Broadview
www.broadview.com.cn

天书夜读

驱网核心技术丛书

天书夜读
——从汇编语言到Windows内核编程

从汇编语言到Windows内核编程

谭文 邵坚磊 著　　罗云彬 审校

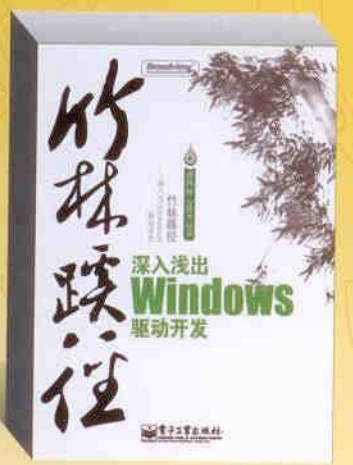
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

驱网核心技术丛书

《天书夜读
——从汇编语言到Windows内核编程》

本书从基本的Windows程序与汇编指令出发，深入浅出地讲解了Windows内核的编程、调试、阅读，以及自行探索的方法。读者在使用C/C++开发Windows程序的基础上，将熟练掌握汇编和C语言的应用，深入了解Windows底层，并掌握阅读Windows内核的基本方法，以及Windows内核的基本编程方法。

本书适合使用C/C++在Windows上编程的读者，尤其适合希望加深自己技术功底的Windows应用程序员、计算机专业的有志于软件开发的大中院校学生；专业的Windows内核程序员，亦可从本书得到超越一般内核程序开发的启发。



《竹林蹊径
——深入浅出Windows驱动开发》

Windows驱动开发的需求越来越多，相关的资料却还不够充实。有时即使读完许多文档，所知依然极为有限。碰到实际的需求、奇怪的错误、莫明的冲突等，依然一筹莫展。本书并不按部就班地介绍Windows驱动体系，而是一切从实践出发，教会读者理解、编写内核代码、调试驱动程序、寻找和修正错误。正如竹林之中蹊径一般，虽然没有大路的宏伟，却能引导入门的读者看穿层层帷幕，随心所欲地实现自己的需求。

《寒江独钓
——Windows内核编程与信息安

几年以前，信息系统虽然在企业中广泛应用，但是信息安全并不为业界所重视。安全软件的开发者，真可谓“独钓寒江雪”。然而时过境迁，到如今，正是大显身手之日！信息安全理论方面的经典著作早已汗牛充栋，但是介绍实际项目开发的书寥寥无几。这是一本少有的，从入门出发直接介绍Windows驱动开发在信息安全项目中的实际应用的书，涉及打印、硬盘、文件、网络方面的Windows驱动开发的细节技术。

本书增值服务

只要您将序列号**B07339**，移动或联通用户发短信至**10666666789**，或者发电子邮件至**market@broadview.com.cn**，即可**获取充足的学习资源**，享受贴心服务。

1. 下载《Windows文件过滤驱动开发教程（第二版）》电子书
2. 下载TDI过滤驱动开发指南

（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

驱网核心技术丛书

从汇编语言到 Windows 内核编程

谭文 邵坚磊 著 罗云彬 审校

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书从基本的 Windows 程序与汇编指令出发，深入浅出地讲解了 Windows 内核的编程、调试、阅读，以及自行探索的方法。读者在使用 C/C++开发 Windows 程序的基础上，将熟练掌握汇编和 C 语言的应用，深入了解 Windows 底层，并掌握阅读 Windows 内核的基本方法，以及 Windows 内核的基本编程方法。

本书适合使用 C/C++在 Windows 上编程的读者，尤其适合希望加深自己技术功底的 Windows 应用程序员、计算机专业的有志于软件开发的大中院校学生；专业的 Windows 内核程序员，亦可从本书得到超越一般内核程序开发的启发。

图书在版编目（CIP）数据

天书夜读：从汇编语言到 Windows 内核编程 / 谭文，邵坚磊著. —北京：电子工业出版社，2008.10
（驱网核心技术丛书）
ISBN 978-7-121-07339-7

Ⅰ. 天… Ⅱ. ①谭… ②邵… Ⅲ. ①汇编语言—程序设计 ②窗口软件，Windows—程序设计
Ⅳ. TP313 TP316.7

中国版本图书馆 CIP 数据核字（2008）第 136007 号

责任编辑：葛 娜
印 刷：北京智力达印刷有限公司
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×980 1/16 印张：17.75 字数：393 千字
印 次：2008 年 10 月第 1 次印刷
印 数：5000 册 定价：45.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

Windows 是庞大复杂的系统。由于 Windows 并不公开源代码，我们在调试程序的时候，往往就调到自己未知的领域去了。没有 C 代码，只能看到令人眼花缭乱的汇编指令和机器码。我曾对它们望而生畏，敬而远之。尤其在这个黑客、破解、病毒、木马横行的时代，如果作为安全软件的开发者，同样不能期盼病毒的作者提供可以阅读的高级语言代码。

如果那些东西，也和 C 语言一样亲切易懂，那多么好啊！这样的话，即便是 Windows 这样庞大复杂而且封闭的系统，或者是再诡异和隐蔽的破坏技术，至少只要我愿意去探索，对我来说就不再有秘密可言。

其实这个梦想并非不切实际。既然我们能读懂 C 代码，何以就不能读懂汇编呢？很多高手眼中，机器指令和 C 代码一样熟悉。

这本书并没有系统地介绍 Windows 系统底层。但是我尝试寻找正确的方法和手段，为读者打开 Windows 底层知识宝库的大门，使读者可以在其中自由阅读，自己去获取所需知识。

本书强调汇编语言的应用。虽然没有像指令手册那样罗列所有指令的细节，但是让读者在实践中熟悉实用的汇编指令。我一般把汇编语言用于调试和理解程序，在写代码时应用较少。但是汇编是了解没有代码的二进制代码的基础。

这里着重于 Windows 内核编程，包括 WDK、WinDbg 的使用，但是 Windows 内核开发体系庞大复杂，这方面已经有很多的经典书籍可以阅读。在这里我只提到了简单、实用和必要的部分。

对于安全软件的开发者，尤其是有兴趣在 Windows 系统下开发反病毒软件的内核驱动读者，会学到基础的研究方法。

由于这一切都从阅读和理解机器指令出发，所以名为“天书夜读”。

“天书夜读”是通向软件底层技术的一个大门与捷径。这里涉及的每个方面的技术，无论是 CPU 底层架构与机器指令、汇编语言、Windows 内核编程、软件逆向工程还是软件调试，无一不是入门艰难的技术，许多技术者在开始时就被汗牛充栋的庞杂资料吓倒。用一条简洁明确的线，把它们串联起来，使读者可以从最基础入门，循序渐进地接触到高深的技术，并应用于实际。避免一开始就面对浩瀚的文档资料而茫然不知如何入手，也不知如何学以致用，这就是我写这本书的目的。

本书正文的内容是从读者很可能已经遗忘的汇编语言的基础指令开始，介绍 C 语言与汇编指令的关系，为阅读用 C 语言编写的 Windows 内核做准备。然后开始讲解 Windows 内核的 C 语言编程的基础。让读者能熟悉使用 C 语言开发 Windows 内核程序。接着指导读者进行内核的开发和调试环境的配置，并进行了一系列的尝试：阅读 Windows 内核的部分实现代码，并尝试自己去实现它。与此同时，产生了修改已有内核的需求。于是接下来介绍了机器码指令和汇编语言的关系，以及能够解读机器码的反汇编引擎，并以此为基础，介绍了进行 Windows 内核 Hook 开发的方法。之后是 Windows 内核 Hook 的一个利用：我们举出了一个防止病毒木马感染的安全防毒软件的例子。在此之后，简要介绍了更深层次的病毒与安全的对抗：Rootkit 与 HIPS。最后是一些防止代码与技术被阅读者“偷学”的章节。

相信读者读完此书，无论是对汇编语言的掌握，还是 Windows 编程的技术，以及对 Windows 系统的了解和调试程序、查找修正故障的能力，都会得到一个飞跃式的提高。

谭　文

2008 年 4 月 2 日

{ — ° ç Ô s < œ

ì {2! > h Ú t X + h t ÿìUì Y'qŁ \ ÿÔhtô! I F'
r Ł¸&ì·ÿp¨Mäh ru ,ÿì pQ'Ip » ÿqÓ{h2%81¸ X
+
§ » öl Ó« ·¸ì ìÿ&ö÷ì O u÷Ñr:hØÒ[vhßO-¸[r:h{òVŒ
I 2ìÞ ßO-ÿ¸[!.3+/ ØÒ[v K [v=l ‹{ {2 2Cÿ{ IÓ¢ 'ÿÕÞ
ì k ¸Àt!I » ÿqÓ{hì FOL @KJAP IB? R>ÿ"\î À I ‡Ò° ÖNâì
Þ» WìFOL @KJAPÀïfflÿ_ w ıý t »
r¸ ÿ ÆH {2 § \t r ! t b <§ h 1ü+$! t b § {Òh
ÿwì'ì=OL JAP * ì ¨y Ü! JAP F AA LDL § h Úh4¶N_ ðl h r
ÿe vy Yu ·ìÿOMHOANRAN ?OO =F=T v ß[> hHtß ìW Öø |l
HtßW"ª I ¸y ˇˇ»+r Wr§ ˇˇ 'æ? ì' æq ? Úæ
I ? ì! ')h § Jh I ') Þs ¥ÿxŁ ðÓÓ b r kk‡à! e
Æ Úh« JAPìÞ˘ hÆF AAÿs7 hÆ=LE{2 hØÒ[v % ‡[y4 ì
ÿ¢ı rŁ¸&ì·ÿMä l » ÿqÓ[! ¸À
´¨ JAP F=R=ÿN_?t Øð!|{ ØÒ[vl ° tÕÞl » !f¸ »ÿh Óh
ý¨t ý !¨h'IB? JAPp‹Ù¶Ö..§ kæÞ hfÿÚh° »k Üt h! '
ràkàt! I » Þt bïÿVŒ t b ÿV 'ì JAPÿ Æ9ÀÆØe Ù¶
÷t÷ð ¥ür ðt b § ltâð Ó° N_ Ót bïÿVŒ Ók Üt
h°ýß E÷ðÿht¨¥ N_ï< h°Ó ÜïÿVŒyWð ÀfÿÚ
Ó[¾ÿ– wJÓßÿ» hß>÷Ñr: O ìÿtÞhx 2Cÿ{ J˘⅜8 xJ
‹SD W`uUÿ÷Ñr: ß>t ÜØÒ[vhxJ Ül N_'ÞÿVŒ ï·k » ØÒ[v
ÿ t Ü|{h{òVŒhx b2CE˘ Ó¸ Ó¸ Øh t { °òÿ–
ß> !ìÞ» ÖNâ Wì JAPhF=R=I #by ÀÚh#s+hï· [ÿ',—¸ h wt°
' þh q hIR? KNII A:hÓhÀð t ý § ðh Úlp Àh Úì Þ Ó«
‹{ h4¶'2hŒß^RI » ìÞ W I k‡ÖtbEpb
¸ tÓ¸ þxæ þ', P ,#2 6+*II',/¨ß¥hk ß> ,ÿ» t ßh ì '
pƒp˘ ÖJhh ! &ht ! Kÿ“ÿ I þ¸ÿ',Û Ł¸F ÿVŒhÓ f
ÿì ',H s Ch ßßÿ«» ïìÞh! I » hÓÕÞe þ', f
Nâ{2E÷òì! ! [v=LE{2h ï· ØFF hì fl· Æ¥ZVŒ ìÞ? !
{2 » h Wì IB? JAPI Às757h wf.ß äŒì 4 {2§Ó hì » 2
4!ÿ2CŒß§Ó> e ß>Óòì4 h 4ì y h tÀ hH >h ^C4hß>òì
y h 4ì4 h ŁÓh§Ó ' ìh“9¸k

{2 Þs k - h ß > ì J A P I B ? R > I þ ¸ ÿ ø Þ ' þ ° - ‰ R h » ÿ ì - h q
‰ R - t ß t k ° ÿ f ¸ ¥ À à fi ´ ¯ Æ b § k ! ô ô h ¯ b
ô ÿ ¸ ‹ 4 ¯ h / ß k - I h ô ¾ 1 o y N â k Õ = ÷ Õ â ÿ I ¢ U ÿ
- h } À ¸ ˙ œ ¯ Y . r k H \ ´ Ó ÿ p ° . - Ñ] H ' y h n k ˘ k ‹ ÿ i
- h æ g k Õ ¯ Æ b ´ Ó s ô ÿ » ÿ h q Ó ÿ Ò t { 2 Þ s ° [h L » Þ !
Ö À H Ô ÿ h w h # ° h ˘ Ò # 8 q I F ï + t ! ð t ° ¢ ÿ t ì h
· m ‡ Õ Þ ÿ »
I ¥ ÿ ; Õ h ü ÿ t f ¥ Z h ¸ k ‡ I & ì · ÿ p ¨ h Ý ó y ¸ Ò À ! ¸ À
ÿ ÿ Ú ¸ h « I & ì · ÿ M ä ÿ ø B ł Œ L @ B ¸ ' s ‹ s 6 2 % ¸ ! 8 1 L @ B
“ › D P P L > > O P D A E P D K I A ? K I N A = @ D P I P E @ D P I H
ü I ¸ “ Ø ¸ Ó Þ ß ÿ » h t Ó h · R { ÿ ' , h Þ » W ì e ',
X . ” ª h / ÿ _ À Þ ÿ
» I q Ó x J ì Þ ï ö l Ú h ì Þ ï · E ÷ É l » 2 [ÿ ÿ ‹ ¸ Ö p ÿ § ì O
Ø { 2 ç – ¸ ö þ ¸ “ o ' + Ø ‡ q + ÿ — – p I I p ÿ § t ¤ 8 1 · Þ » h w
· ÿ 2 C ¥ ï ï ‡ ¸ ÿ _ h Ý à 6 ó l § y h · ÿ 2 C + À = † ı ¨ I
§ ? t Þ ! î \ h ¥ ÿ ¸ ì ð l \ ô 0 ÿ t ¸ I Y å { Œ Ø , ! 4 ô À
! ^ ! M Ö Ø ‡ h ¸ œ w Æ Ø ì I § h À ì ÿ . ß Ô ! % I J
‡ y ÿ ì s ß ÿ »
+ Ø ‡ × § j > > O P D A E P D K I A ? K I
ß O - ì » t — ß ÿ " \ ¨ ÿ { h Ó q -
ì Ú † p ÿ § h ¨ E § h x J k -
ì Ý × Ö ÿ “ ÿ h ì Þ h t À e ^
q + B Æ Ø h v h À ì ÿ . ß Ô
Æ Ø ÿ k p ÿ § h q Ö + k À â · a î q + Æ Ø f ! ÿ

目　录

入手篇　熟悉汇编

本书的第一部分，将帮助读者消除对汇编的恐惧，熟悉汇编。本部分包括第 1~3 章。稍显枯燥的是，它们和 Windows 内核无关，是纯 C 语言与汇编语言的关系的章节。如果读者已经精通汇编语言，并能顺利阅读汇编代码，请直接跳过本部分。

基础篇 内核编程

本书的第二部分，是编写 Windows 内核程序编程方法的基础。本部分包括第 4~7 章，如果读者对 Windows 内核编程已经有一定的了解，可以跳过本部分；如果读者从未接触过 Windows 内核编程，本部分将指导读者开始 Windows 内核编程，学会使用 WDK，并熟悉内核编程的习惯与方法。

探索篇 研究内核

本书的第三部分，开始探索 Windows 内核程序，并尝试阅读反汇编代码作为指引。本部分包括第 8~10 章。如果读者对 Windows 内核编程已经有一定的了解，这一部分会比较有趣；如果读者从未接触过 Windows 内核编程，读者应该先学习第二部分。能自己编写内核程序并不意味着可以读懂内核，虽然反过来是一定成立的。读懂别人编写的没有代码的程序，比自己编写更困难一些，但的确是值得的。

深入篇　修改内核

这是本书的第四部分。读者已经尝试过探索 Windows 内核程序，并尝试阅读反汇编代码。那么接下来，必须掌握修改内核的方法。每一个 Windows 内核程序，都可以看做 Windows 内核本身的一个“补丁”。有时只需要独立存在，就能起到它的作用；有时却必须对已有的内核二进制代码进行部分修改。本部分包括第 11~13 章，主要介绍的是内核 Hook。

实战篇　实际开发

实战部分是本书最深入和复杂的一部分，包括第 14～17 章。为了让前面练习的成果，在实际应用中产生价值，在这部分我们补充更多的理论知识并尝试用它们去做一点什么。这一部分包括指令分析、硬件基础知识、内核 Hook 的实际开发练习，以及将完成一个用到内核 Hook 的有趣的实例，这个实例有助于计算机阻挡各种病毒和木马的侵袭。

此外，本部分还包括特殊的一章，涉及如何巧妙地编写代码，来防止被其他不受欢迎的读者阅读。这与本书的主旨完全相反，正所谓物极必反。

入手篇

熟悉汇编

本书的第一部分，将帮助读者消除对汇编的恐惧，熟悉汇编。本部分包括第 1~3 章。稍显枯燥的是，它们和 Windows 内核无关，是纯 C 语言与汇编语言的关系的章节。如果读者已经精通汇编语言，并能顺利阅读汇编代码，请直接跳过本部分。

第1章　汇编指令与C语言

本书总是假设读者的“母语”是 C/C++。（如果是 Java 或者 C#，建议读者首先学习 C 语言。）由于 Microsoft 并没有在 Windows 中附带所有的 C 代码，因此读者遇到的很多将是汇编代码。

那么理解汇编代码就成为当前最大的问题。汇编代码之所以看不懂，除了汇编代码可读性本来就比较差之外，更重要的一点原因是，读者的母语是 C 语言。甚至会常常觉得，自己虽然能看懂每一行指令，却完全看不懂一段程序在做什么。

但是实际上，Windows 的内核代码基本上都是由 C 语言代码编译而成的。这些生成的机器码，再被调试器反汇编得到的汇编语言，和 C 语言有着千丝万缕的联系。只要熟悉它们之间的对应关系，像理解 C 语言一样理解汇编代码，就完全是可能的了。

为了强化这种关系，本书有时从汇编代码写出对应的 C 语言，以便于读者理解。

何为软件反工程（逆向工程）

作为编程爱好者、程序员，大多数人从事“软件工程”的工作。这种工作是从人类可以理解的高级语言编译为机器可以理解的机器码的过程。反工程则是相反的，是从机器可以理解的机器码，到人类可以理解的某种形式的过程。其目的在于，通过别人出售的产品得到别人的工程技术。

反工程往往被看作是旁门左道。因此在正式出版的计算机书籍中提及并不多，可谓讳莫如深；而破解者、黑客、病毒制造者则对此乐此不疲。反工程有专门的软件工具、知识体系，许多“秘笈”在网络上流传。系统的反工程技术涉及从还原下层逻辑一直到大型软件的顶级架构。

这些都不是本书的内容，也不是作者有能力讲述的范围。本书仅仅让读者学会阅读调试器中的汇编代码，以加深对 Windows 底层的理解，增加读者解决下层 bug 的能力。对反工程没有涉及。

请注意，对软件的反工程行为，如果未取得合法的授权，则是侵犯知识产权的非法行为。

有兴趣了解反工程的读者，可以阅读《Reversing:逆向工程解密》。此书信息如下：

英文名：Reversing:Secrets of Reverse Engineering

中文名：Reversing：逆向工程解密

作者：　Eldad Eilam;Elliot Chikofsky

译者：　韩琪、杨艳等

出版社：电子工业出版社

不过，回归主题，建议读者还是先继续阅读本书，继续做好 Windows 底层程序员这份有前途的职业吧。

1.1　上机建立第一个工程

一条指令可能有很多细节，比如第二个操作数和第一个操作数中，哪个只能是寄存器，哪个不能用寄存器加立即数等。如果要用手写汇编语言来开发一个软件的话，这些知识很重要。但是实际上，现在需要的是用 C 语言或者更高级的语言进行开发，只是需要理解一些没有 C 代码的部分。读者可以认为，那些代码的汇编指令用法一定是正确的，因为它们是编译器生成的。

1.2 节（简要复习常用的汇编指令）将简要地回顾那些指令，并保证这些回顾非常容易理解。

1.1.1　用 Visual Studio 创建工程

建议读者的计算机中操作系统的版本最好是 Windows XP 或者更高的版本，使用 Vista 问题也不大，虽然可能操作细节和界面与本书描述的情况稍有不同。然后，请安装某个版本的 Visual Studio，推荐 Visual Studio 2003 或者更高的版本。下面的描述同时顾及 Visual Studio 2003 和 Visual Studio 2005 的情况，语言版本为英文。

下面的步骤将在 Visual Studio 上建立一个用于实验的工程。有经验的读者可以忽略下面的描述。

1 打开 Microsoft Visual Studio 2005，选择主菜单“File”。

2 选择子菜单“New”下面的“Project”，打开“New Project”对话框。

3 左边选择 Visual C++，Win32；右边选择 Win32 Console Application；下面输入一个工程名，然后单击“OK”按钮，出现向导。一切都选择默认设置，最后单击“Finish”按钮。

此时新建立工程的主文件是一个扩展名为.cpp 的文件，请将它改为扩展名为.c 的文件。

Windows 底层代码基本上都是用 C 语言编写的，研究 C 语言和汇编指令的关系是本节的任务。为此，这个文件必须改变为扩展名为.c 的文件，这样 VC 才会自动以 C 语言的方式进行编译。本书后面专门有一章的内容来研究 C++下的反汇编阅读（第 9 章 用 C++编写的内核程序）。

4 如果读者使用的是 Visual Studio 2005，直接在左边的 Solution Explorer 中用鼠标右键单击文件 YourProjectName.cpp，选择“Rename”，然后把.cpp 改为.c 即可。

如果是更老的 Visual Studio 版本，请单击右键，Remove 这个文件。然后在外面改名，再对工程单击右键，选择“Add”，再选择“Exist Item”，追加进来。

此时程序是这样的：

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
        return 0;
}
```

1.1.2 用 Visual Studio 查看汇编代码

C 语言程序对应的汇编代码，可以在 VC 中非常清楚地显示出对应关系。但是并不是所有的读者都知道如何调出汇编指令窗口。这个诀窍在下面描述。

1 VC 必须处于调试状态才能看到汇编指令窗口。因此，请在 return 0 一句上设置一个断点：把光标移到那一行，然后按下 F9 键设置一个断点。

2 按下 F5 键调试程序。当程序停止在这一行的时候，打开菜单“Debug”下的“Windows”子菜单，选择“Disassembly”。这样，出现一个窗口，显示下面的信息：

```
--- f:\root\work\any\t12\t12\t12.c ----------
// t12.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
00411360  push    ebp
00411361  mov     ebp,esp
00411363  sub     esp,0C0h
```

```
00411369  push    ebx
0041136A  push    esi
0041136B  push    edi
0041136C  lea     edi,[ebp-0C0h]
00411372  mov     ecx,30h
00411377  mov     eax,0CCCCCCCCh
0041137C  rep stos   dword ptr es:[edi]
    return 0;
0041137E  xor     eax,eax
}
00411380  pop     edi
00411381  pop     esi
00411382  pop     ebx
00411383  mov     esp,ebp
00411385  pop     ebp
00411386  ret
```

如果上面的内容完全看不懂，也许读者需要复习一下汇编指令。以上的汇编指令数量非常的少，只需要了解 push、mov、sub、lea、stos、xor、pop、ret，就可以继续本书的学习之旅了。所以请不用担心，接下来就会熟悉这些代码。

1.2 简要复习常用的汇编指令

1.2.1 堆栈相关指令

对指令的详细了解，应该查阅 Intel 发布的指令手册。但是从头开始阅读那些手册是一件令人望而生畏的事情。读者暂时可以只初步了解，忽略一些细节，当实际用到的时候，再具体查阅相关指令。

- push：把一个 32 位的操作数压入堆栈中。这个操作导致 esp 被减 4。esp 被形象地称为栈顶。我们认为顶部是地址小的区域，那么，压入堆栈的数据越多，这个堆栈也就越堆越高，esp 也就越来越小。在 32 位平台上，esp 每次减少 4（字节）。
- pop：相反，esp 被加 4，一个数据出栈。pop 的参数一般是一个寄存器，栈顶的数据被弹出到这个寄存器中。

一般不会把 sub、add 这样的算术指令，以及 call、ret 这样的跳转指令归入堆栈相关指令中。但是实际上在函数参数传递过程中，sub 和 add 最常用来操作堆栈；call 和

ret 对堆栈也有影响。所以这里做特殊处理。

- sub：减法。第一个参数是被减数所在的寄存器；第二个参数是减数。（对应的还有 add 指令。）
- add：加法。
- ret：返回。相当于跳转回调用函数的地方。（对应的 call 指令来调用函数，返回到 call 之后的下一条指令。）
- call：调用函数。

说到这里，有必要详述一些指令对堆栈的影响。某些指令会“自动”地操作堆栈，这就是 call 和 jmp 的不同之处。call 指令会把它的下一条指令的地址压入堆栈中，然后跳转到它调用的函数的开头处；而单纯的 jmp 是不会这样做的。同时，ret 会自动地弹出返回地址。

call 的本质相当于 push+jmp。ret 的本质相当于 pop+jmp。

不但 push、pop、call 和 ret 会操作堆栈，sub 和 add 也可以用于操作堆栈。如果我要一次在堆栈中分配 4 个 4 字节长整型的空间，那么没有必要 4 次调用 push，很简单地把 esp 减去 4*4=16 即可。当然，也可以同样地用 add 指令来恢复它。这常常用于分配函数局部变量空间，因为 C 语言函数的局部变量保存在栈里。

1.2.2　数据传送指令

- mov：数据移动。第一个参数是目的，第二个参数是来源。在 C 语言中相当于赋值号。这是最广为人知的指令。
- xor：异或。这虽然是逻辑运算的指令，但是有趣的是，xor eax,eax 这样的操作常常用来代替 mov eax,0。好处是速度更快，占用字节数更少。
- lea：取得地址（第二个参数）后放入到前面的寄存器（第一个参数）中。

见到 xor eax,eax，应该马上明白这是清零操作。

但是实际上，有时候 lea 用来做和 mov 同样的事情，比如赋值。看下面一条指令：

```
lea edi,[ebp-0cch]
```

方括弧表示存储器，也就是 ebp-0cch 这个地址所指的存储器内容。但是 lea 要求取[ebp-0cch]的地址，那么地址也就是 ebp-0cch，这个地址将被放入到 edi 中。换句话说，这等同于：

```
mov edi,ebp-0cch
```

但是以上 mov 指令是错误的，因为 mov 不支持后一个操作数写成寄存器减去数字，但是 lea 支持，所以可以用 lea 来代替它。

指令的操作数能采用的运算符号有非常复杂的限制。如果需要使用，应该查询指令手册。

为了讲解 stos，下面解说前面提到的代码：

```
mov     ecx,30h
mov     eax,0CCCCCCCCh
rep     stos dword ptr es:[edi]
```

stos 是串存储指令，它的功能是将 eax 中的数据放入 edi 所指的地址中，同时，edi 会增加 4（字节数）。rep 使指令重复执行 ecx 中填写的次数。方括弧表示存储器，这个地址实际上就是 edi 的内容所指向的地址。这里的 stos 其实对应的是 stosd，其他还有 stosb、stosw，分别对应于处理 4、1、2 个字节，这里对堆栈中 30h*4(0c0h)个字节初始化为 0cch（也就是 int 3 指令的机器码），这样发生意外时执行堆栈里面的内容会引发调试中断。

1.2.3 跳转与比较指令

部分跳转指令如下。

- jmp：无条件跳转。这也是多年后我依然未忘记的少量指令之一。
- jg：顾名思义，大于的时候跳转。通常前面有一条比较指令。
- jl：顾名思义，小于的时候跳转。通常前面有一条比较指令。
- jge：顾名思义，大于等于的时候跳转。通常前面有一条比较指令。

类似的指令还有一些，这里就不介绍了。

下面介绍一条比较指令。

- cmp：顾名思义，比较。往往是 jg、jl、jge 之类的条件跳转指令的执行条件。

本书在这里只是为了满足阅读后面章节的需要，简单地介绍了部分汇编指令，一些后面才会碰到的指令则在遇到时再专门提及。对于希望进一步学习汇编语言的读者，我推荐《Windows 环境下的 32 位汇编语言程序设计》一书。这本书的信息如下：

> 书名：Windows 环境下的 32 位汇编语言程序设计
>
> 作者：罗云彬
>
> 出版社：电子工业出版社

相对于复杂艰深的 Intel 指令手册，此书会比较容易入手。

1.3　C 函数的参数传递过程

基础知识

函数调用的本质将在这里得到阐明。首先请读者理解堆栈的操作。函数和堆栈的关系密切，这是因为：C 语言程序通过堆栈把参数从函数外部传入到函数内部。此外，在堆栈中划分区域来容纳函数的内部变量。

调用 push 和 pop 指令的时候，寄存器 esp 用于指向栈顶的位置——栈顶总是栈中地址最小的位置。push 执行的结果，esp 总是减少，pop 则增加。对于 C 程序默认的调用方式，堆栈总是调用方把参数反序（从右到左）地压入堆栈中，被调用方把堆栈复原（这些我们会在后面见到）。这些参数对齐到机器字长，16 位、32 位、64 位 CPU 下分别对齐到 2、4、8 个字节。这种调用是 C 编译器默认的 C 方式。

函数调用规则

在一个编写高级语言的程序员的观念中，函数（或者没有返回值的过程）是必不可少的基础单元。C 语言的程序完全由函数构成，所有的代码都在某一个函数中。Pascal 区分函数和过程，但是本质依然是类似的。对计算机硬件而言，这种区分毫无必要，因为 CPU 只关心一条一条的指令，并不关心它们是以怎样的结构组织的。

call 指令和 ret 指令只是为了调用的方便而已，绝不是函数存在的绝对证据。即

使我们仅仅使用 jmp 并自己操作堆栈，也一样可以实现函数的功能。因此，一种高级语言如何实现函数调用，并没有法律的约束，所以出现了各种不同的函数调用规则。

但是毫无疑问，如果一个第三方提供的函数要能被使用，那么必须有约定的函数调用规则。

函数调用规则指的是调用者和被调用函数间传递参数及返回参数的方法，在 Windows 上，常用的有 Pascal 方式、WINAPI 方式（_stdcall）、C 方式（_cdecl）。

_cdecl C 调用规则：

（1）参数从右到左进入堆栈；

（2）在函数返回后，调用者要负责清除堆栈，所以这种调用常会生成较大的可执行程序。

_stdcall 又称为 WINAPI，其调用规则：

（1）参数从右到左进入堆栈；

（2）被调用的函数在返回前自行清理堆栈，所以生成的代码比 cdecl 小。

Pascal 调用规则：

Pascal 调用规则主要用在 Win16 函数库中，现在基本不用。

（1）参数从左到右进入堆栈；

（2）被调用的函数在返回前自行清理堆栈。

（3）不支持可变参数的函数调用。

此外，在 Windows 内核中还常见有快速调用方式（_fastcall）；在 C++编译的代码中有 this call 方式（_thiscall）。这些会在后面的章节中详细阐明。

技术细节

在用 C 语言所写的程序中，堆栈用于传递函数参数。写一个简单的函数如下：

```
void myfunction(int a,int b)
{
    int c = a+b;
}
```

这是标准的 C 函数调用方式。其过程是：

1 调用者把参数反序地压入堆栈中。

2 调用函数。

3 调用者把堆栈清理复原。

这就是 C 编译器默认的_cdecl 方式，而 Windows API 一般采用的_stdcall 则是被调用者恢复堆栈（可变参数函数调用除外）。

至于返回值都是写入 eax 中，然后返回的。

在 Windows 中，不管哪种调用方式都是返回值放在 eax 中，然后返回。外部从 eax 中得到返回值。

_cdecl 方式下被调用函数需要做以下一些事情。

（1）保存 ebp。ebp 总是被我们用来保存这个函数执行之前的 esp 的值。执行完毕之后，我们用 ebp 恢复 esp；同时，调用此函数的上层函数也用 ebp 做同样的事情。所以先把 ebp 压入堆栈，返回之前弹出，避免 ebp 被我们改动。

（2）保存 esp 到 ebp 中。

上面两步的代码如下：

```
;保存 ebp，并把 esp 放入 ebp 中，此时 ebp 与 esp 同
;都是这次函数调用时的栈顶
push ebp
mov ebp,esp
```

（3）在堆栈中腾出一个区域用来保存局部变量，这就是常说的所谓局部变量是保存在栈空间中的。方法是：把 esp 减少一个数值，这样就等于压入了一堆变量。要恢复时，只要把 esp 恢复成 ebp 中保存的数据就可以了。

（4）保存 ebx、esi、edi 到堆栈中，函数调用完后恢复。

对应的代码如下：

```
;把 esp 往下移动一个范围，等于在堆栈中放出一片新
;的空间用来存局部变量
sub esp,0cch
push ebx     ;下面保存三个寄存器：ebx、esi、edi
push esi
```

```
push edi
```

（5）把局部变量区域初始化成全 0cccccccch。0cch 实际是 int 3 指令的机器码，这是一个断点中断指令。因为局部变量不可能被执行，如果执行了，必然程序有错，这时发生中断来提示开发者。这是 VC 编译 Debug 版本的特有操作。相关代码如下：

```
lea edi,[ebp-0cch]    ;本来是要 mov edi,ebp-0cch，但是 mov 不支持
                      ;ebp-0cch 这样的参数
                      ;所以对 ebp-0cch 取内容，而 lea 把内容的地址，也就
                      ;是 ebp-0cch 加载到 edi 中。目的是把保存局部变量
                      ;的区域（从 ebp-0cch 开始的区域）初始化成全
                      ;部 0cccccccch
mov ecx,33h
mov eax,0cccccccch
rep stos dword ptr [edi] ;串写入
```

（6）然后做函数里应该做的事情。参数的获取是 ebp+12 字节为第二个参数，ebp+8 为第一个参数（注意倒序压入），依次增加。最后 ebp+4 字节处是要返回的地址。

（7）恢复 ebx、esi、edi、esp、ebp，最后返回。代码如下：

```
pop edi             ;恢复 edi、esi、ebx
pop esi
pop ebx
mov esp,ebp         ;恢复原来的 ebp 和 esp，让上一个调
                    ;用的函数正常使用
pop ebp
ret
```

为了简单起见，我的函数没有返回值。如果要返回值，函数应该在返回之前，把返回值放入 eax 中。外部通过 eax 得到返回值。

代码分析

用 VC 2003 编译 Debug 版本，完整的反汇编代码如下：

```
void myfunction(int a,int b)
{
push ebp      ;保存 ebp，并把 esp 放入 ebp 中。此时 ebp 与 esp 同
mov ebp,esp   ;都是这次函数调用时的栈顶
sub esp,0cch  ;把 esp 往上移动一个范围，等于在堆栈中放出一片新
              ;的空间用来存储局部变量
push ebx      ;下面保存三个寄存器：ebx、esi、edi
push esi
```

```
push edi
lea edi,[ebp-0cch]    ;本来是要“mov edi,ebp-0cch”，但是mov不支持
                      ;“-”操作，所以对ebp-0cch取内容，而lea把内容
                      ;的地址，也就是ebp-0cch加载到edi中。目的是
                      ;把保存局部变量的区域（从ebp-0cch开始的区域）
                      ;初始化成全部0cccccccch
mov ecx,33h
mov eax,0cccccccch
rep stos dword ptr [edi] ;写入0cch指令（中断）
    int c = a+b;
mov eax,dword ptr [a]     ;简单的相加操作。这里从堆栈中取得从外部
                          ;传入的参数。那么，a和b到底是怎么取得的呢
    add eax,dword ptr[b] ;通过ida反汇
                          ;编可以看到，其实这两条指令是
                          ;mov eax, [ebp+8], add eax, [ebp+0Ch]
                          ;参数是通过ebp从堆栈中取得的。这里看到
                          ;的是VC调试器的显示结果，为了阅读方
                          ;便，直接加上了参数名
mov dword ptr[c],eax
}
pop edi                   ;恢复edi、esi、ebx
pop esi
pop ebx
mov esp,ebp               ;恢复原来的ebp和esp，让上一个调用的函数
                          ;正常使用
pop ebp
ret
```

主程序中对这个函数的调用方式是：

```
mov eax,dword ptr[b]      ;把b、a两个参数压入堆栈
push eax
mov ecx,dword ptr[a]
push ecx
call myfunction           ;调用函数myfunction
add esp,8                 ;恢复堆栈
```

这样一来，函数调用的过程就很清楚了。在下一章开始，进一步介绍各种各样的 C 语言程序，变成了怎样的汇编指令。

重点观察那些涉及 call、ret、push 和 pop，操作 ebp 和 esp 的指令，就能看到 C 语言函数的调用过程。

第2章　C语言的流程和处理

2.1　C 语言的循环反汇编

为了简单起见，在这一章的内容中我们始终只研究调试版本的反汇编结果。后面的章节再专门研究发行版本的反汇编。本节研究循环流程。请注意，虽然在这里能看到一些规律，但是这里的结果并不是一成不变的。在实际应用中，要考虑到各种可能的变化。

何为 Debug（调试版）和 Release（发行版）

在高级语言出现的早期，很可能根本就没有这个区别。高级语言被编译成可执行的机器码，一种版本就足够了。但是很快人们发现那些可执行的机器码是可以被优化的。比如说已经定义了但是没有被使用的变量、根本不可能执行到的部分代码，完全可以去除。一些跳转来跳转去的复杂逻辑，可以用简单的跳转替代。一些多条指令才达到目的的情况，实际上用一条指令就足够了。连续不断的优化操作，使最终编译结果越来越简洁、高效。

但是这又出现了另一个问题。当 bug 发生时，开发者需要调试。然而经过优化的代码，常常看上去已经和原来的高级语言逻辑大相径庭（虽然它们之间的功能应该是等价的）。这样一来调试者就往往摸不着头脑。此时就希望编译器不要优化或者少优化，于是出现所谓“调试版”和“发行版”的区别。调试版基本不优化；发行版则一般都优化到极致。

在 Windows 应用程序编程中，一般称为 Debug 版本和 Release 版本。但是这个名字并非放之四海而皆准，比如 Windows 驱动开发中，又一般称为 Check 版本和 Free 版本。但是意义是一样的。

2.1.1　for 循环

下面把函数改得复杂一点，增加一个循环，来看看反汇编的结果。

```
int myfunction(int a,int b)
{
    int c = a+b;
    int i;
```

```
    for(i=0;i<50;i++)
    {
        c = c+i;
    }
    return c;
}
```

开始部分的反汇编结果和 1.3 节是一致的，现在从 for 的地方开始反汇编，结果如下：

```
for(i=0;i<50;i++)
00412BC7  mov     dword ptr [i],0
00412BCE  jmp     myfunction+39h (412BD9h)
00412BD0  mov     eax,dword ptr [i]
00412BD3  add     eax,1
00412BD6  mov     dword ptr [i],eax
00412BD9  cmp     dword ptr [i],32h
00412BDD  jge   myfunction+4Ah (412BEAh)
{
    c = c+i;
00412BDF  mov     eax,dword ptr [c]
00412BE2  add     eax,dword ptr [i]
00412BE5  mov     dword ptr [c],eax
}
00412BE8  jmp     myfunction+30h (412BD0h)
00412BEA  mov     eax,dword ptr [c]
```

后面省略的部分和 1.3 节相同。可以看到循环主要用这么几条指令来实现：mov 进行初始化；jmp 跳过修改循环变量的代码；cmp 实现条件判断；jge 根据条件跳转。用 jmp 回到修改循环变量的代码进行下一次循环。所以结构大体如下：

```
mov <循环变量>,<初始值>          ;给循环变量赋初值
jmp B                            ;跳到第一次循环处
A:  (改动循环变量)               ;修改循环变量
    …
B:  cmp <循环变量>,<限制变量>    ;检查循环条件
    jge 跳出循环
    (循环体)
    …
    jmp A                        ;跳回去修改循环变量
```

2.1.2 do 循环

再看一下 do 循环，因为 do 循环没有修改循环变量的部分，所以比 for 循环要简单

一些。

```
    do {
        c = c+i;
00411A55  mov         eax,dword ptr [c]
00411A58  add         eax,dword ptr [i]
00411A5B  mov         dword ptr [c],eax
    } while(c< 100);
    00411A5E  cmp     dword ptr [c],64h
    00411A62  jl      myfunction+35h (411A55h)
    return c;
    …
```

上面的 do 循环就是用一个简单的条件比较指令跳转回去。只有两条指令：

```
cmp <循环变量>,<限制变量>
jl  <循环开始点>
```

2.1.3　while 循环

下面看看 while 循环的情况。

```
        while(c<100){
00411A55  cmp         dword ptr [c],64h
00411A59  jge         myfunction+46h (411A66h)
            c = c+i;
00411A5B  mov         eax,dword ptr [c]
00411A5E  add         eax,dword ptr [i]
00411A61  mov         dword ptr [c],eax
        }
    00411A64  jmp         myfunction+35h (411A55h) return c;
```

读者会发现 while 要更复杂一点。因为 while 除了开始的时候判断循环条件之外，后面还必须有一条无条件跳转回到循环开始的地方，共用 3 条指令实现：

```
A:  cmp <循环变量>,<限制变量>
    jge  B
    (循环体)
    …
    jmp A
B:  (循环结束了)
```

2.2 C 语言判断与分支的反汇编

代码分析

2.2.1 if-else 判断分支

写一个简单的 if 判断结构：

```
if(c>0 && c<10)
{
    printf("c>0");
}
else if( c>10 && c<100)
{
    printf("c>10 && c<100");
}
else
{
    printf("c>10 && c<100");
}
```

if 判断都是使用 cmp 再加上条件跳转指令。对于 if(A && B)的情况，一般都是使用否决法。如果 A 不成立，立刻跳到下一个分支。依次，如果 B 不成立，同样跳到下一个分支。从这里看出，在用 C 语言写一系列条件比较的时候，应该注意排列这些比较，让比较次数最少为好。

```
cmp <条件>
jle <下一个分支>
```

所以开始部分的反汇编为：

```
    if(c>0 && c<10)
00411A66  cmp     dword ptr [c],0
00411A6A  jle     myfunction+61h (411A81h)      ;跳到下一个 else if 的
                                                ;判断点
00411A6C  cmp     dword ptr [c],0Ah
00411A70  jge     myfunction+61h (411A81h)      ;跳到下一个 else if 的
                                                ;判断点
{
```

```
    printf("c>0");
00411A72    push    offset string "c>0" (4240DCh)
00411A77    call    @ILT+1300(_printf) (411519h)
00411A7C    add     esp,4
}
```

else if 和 else 的特点是，在开始的地方，都有一条无条件跳转指令，跳转到判断结束处，阻止前面的分支执行结束后，直接进入这个分支的可能。这个分支能执行到的唯一途径是前面的判断条件不满足。

else 则在 jmp 之后直接执行操作。而 else if 则开始重复 if 之后的操作，用 cmp 比较，然后用条件跳转指令进行跳转。

```
else if( c>10 && c<100)
00411A7F  jmp     myfunction+89h (411AA9h)          ;直接跳到判断块
                                                    ;外

00411A81  cmp     dword ptr [c],0Ah                 ;比较+条件跳转
                                                    ;目标为下一个分
                                                    ;支处
00411A85  jle     myfunction+7Ch (411A9Ch)
00411A87  cmp     dword ptr [c],64h
00411A8B  jge     myfunction+7Ch (411A9Ch)
```

2.2.2 switch-case 判断分支

在条件分支语句中，情况比较特殊的是 switch。switch 的特点是有多个判断。因为 switch 显然不用判断大于小于，所以都是 je，分别跳到每个 case 处。最后一个是无条件跳转，直接跳到 default 处。如以下的代码：

```
switch(c)
{
case 0:
    printf("c>0");
case 1:
    {
        printf("c>10 && c<100");
        break;
    }
default:
    printf("c>10 && c<100");
}
```

反汇编的结果是：

```
switch(c)
00411A66  mov     eax,dword ptr [c]
00411A69  mov     dword ptr [ebp-0E8h],eax
00411A6F  cmp     dword ptr [ebp-0E8h],0
00411A76  je      myfunction+63h (411A83h)
00411A78  cmp     dword ptr [ebp-0E8h],1
00411A7F  je      myfunction+70h (411A90h)
00411A81  jmp     myfunction+7Fh (411A9Fh)
{
    …
```

连续的比较与条件跳转让人联想到 switch。

上面的代码显然是在比较 c 是否是 0、1 这两个数字。至于先把 c 移动到 ebp-0e8h 这个地址，然后再比较，这是调试版本编译的特点。目的并不是很明确。最后一条直接跳转到 default 处，如果没有 default，就跳到 switch 之外。

至于 case 和 default 都非常简单。如果有 break，则会增加一个无条件跳转；在没有 break 的情况下，没有任何流程控制代码。

```
case 0:
    printf("c>0");
00411A83  push    offset string "c>0" (4240DCh)
00411A88  call    @ILT+1300(_printf) (411519h)
00411A8D  add     esp,4
case 1:
    {
        printf("c>10 && c<100");
00411A90  push    offset string "c>10 && c<100" (424288h)
00411A95  call    @ILT+1300(_printf) (411519h)
00411A9A  add     esp,4
        break;
00411A9D  jmp     myfunction+8Ch (411AACh)
    }
default:
    printf("c>10 && c<100");
00411A9F  push    offset string "c>10 && c<100" (424288h)
00411AA4  call    @ILT+1300(_printf) (411519h)
00411AA9  add     esp,4
}
```

思考与练习

下面做一个练习，内容是把下面的汇编代码还原成 C 语言。这种练习非常有好处，请务必一开始就勤加练习。这是消除对汇编的陌生感，顺利完成本节学习目的的唯一方法。

以 C 语言为母语的程序员，常常找一些汇编语言改写为 C 语言，是学习汇编的快捷方式之一。

```
00411A20  push    ebp
00411A21  mov     ebp,esp
00411A23  sub     esp,0E8h
00411A29  push    ebx
00411A2A  push    esi
00411A2B  push    edi
00411A2C  lea     edi,[ebp-0E8h]
00411A32  mov     ecx,3Ah
00411A37  mov     eax,0CCCCCCCCh
00411A3C  rep stos  dword ptr [edi]
00411A3E  mov     eax,dword ptr [a]
00411A41  add     eax,dword ptr [b]
00411A44  mov     dword ptr [d],eax
00411A47  mov     dword ptr [i],1
00411A4E  mov     dword ptr [c],0
00411A55  cmp     dword ptr [c],64h
00411A59  jge     myfunction+46h (411A66h)
00411A5B  mov     eax,dword ptr [c]
00411A5E  add     eax,dword ptr [i]
00411A61  mov     dword ptr [c],eax
00411A64  jmp     myfunction+35h (411A55h)
00411A66  mov     eax,dword ptr [c]
00411A69  mov     dword ptr [ebp-0E8h],eax
00411A6F  cmp     dword ptr [ebp-0E8h],0
00411A76  je      myfunction+63h (411A83h)
00411A78  cmp     dword ptr [ebp-0E8h],1
00411A7F  je      myfunction+6Ah (411A8Ah)
00411A81  jmp     myfunction+72h (411A92h)
00411A83  mov     dword ptr [d],1
00411A8A  mov     eax,dword ptr [c]
00411A8D  mov     dword ptr [d],eax
00411A90  jmp     myfunction+79h (411A99h)
```

```
00411A92  mov     dword ptr [d],0
00411A99  mov     eax,dword ptr [d]
00411A9C  pop     edi
00411A9D  pop     esi
00411A9E  pop     ebx
00411A9F  mov     esp,ebp
00411AA1  pop     ebp
00411AA2  ret
```

2.3 C 语言的数组与结构

下面来写一个用到了结构体和数组的简单函数。

```
typedef struct {
    int a;
    int b;
    int c;
} mystruct;
int myfunction(int a,int b)
{
    unsigned char *buf[100];
    mystruct *strs = (mystruct *)buf;
    int i;
    for(i=0;i<5;i++)
    {
        strs[i].a = 0;
        strs[i].b = 1;
        strs[i].c = 2;
    }
    return 0;
}
```

对结构体和数组的访问是我们感兴趣的地方。相关的反汇编结果是这样的：

```
int i;
for(i=0;i<5;i++)
0041367A  mov     dword ptr [i],0          ;典型的 for 循环过程
00413684  jmp     myfunction+45h (413695h)
00413686  mov     eax,dword ptr [i]
0041368C  add     eax,1
0041368F  mov     dword ptr [i],eax
```

```
00413695  cmp     dword ptr [i],5
0041369C  jge     myfunction+94h (4136E4h)
{
    strs[i].a = 0;
0041369E  mov     eax,dword ptr [i]          ;目的是把 i*0ch 放入到
                                             ;eax 中
004136A4  imul    eax,eax,0Ch                ;0Ch 是结构的大小
004136A7  mov     ecx,dword ptr [strs]       ;把 strs 的地址放入 ecx
004136AD  mov     dword ptr [ecx+eax],0      ;计算得到 strs[i]的地址
                                             ;并赋 0
    strs[i].b = 1;
004136B4  mov     eax,dword ptr [i]
004136BA  imul    eax,eax,0Ch
004136BD  mov     ecx,dword ptr [strs]

;这里与.a 不同，增加偏移取得 b 的位置
004136C3  mov     dword ptr [ecx+eax+4],1
    strs[i].c = 2;
004136CB  mov     eax,dword ptr [i]
004136D1  imul    eax,eax,0Ch
004136D4  mov     ecx,dword ptr [strs]
004136DA  mov     dword ptr [ecx+eax+8],2
}
004136E2  jmp     myfunction+36h (413686h) ;典型的循环结束
004136E4  xor     eax,eax                  ;eax 清零
...
```

imul 指令让人联想到结构体数组，这是一个特征。程序在访问一个结构体的数组时，往往需要得到数组中某个结构体元素的开始地址。很显然，某个元素的起始地址=下标×单个元素的长度+数组的起始地址。单个元素的长度在反汇编代码中往往是一个常数，如上面代码中出现的 0Ch，这个数字由编译器生成。此后，程序中会出现用 imul 指令，将元素下标去乘这个常数的代码。

从这些代码中能比较容易地看出一个结构体数组的存在，并获得结构体的大小。

访问结构体的第一个成员当然不需要增加任何偏移，之后访问第 n 个成员的时候，会加上常数偏移，如上面成员 b 所在的位置是 4（字节）。

在理解反汇编的时候，这些常数很重要。因为许多系统中结构的大小和成员变量所在的偏移是已知的（根据 WDK 开发包中提供的数据结构，详情请见第 4 章开始的内核编程的章节），而那些重要成员所在的偏移就成了一个个让人熟悉的数字。

反汇编代码访问结构体时，结构体的大小和成员变量在结构体中的偏移都是以常数表示的，有时根据这些常数就能直接联想到这段代码操作的结构体。

2.4 C 语言的共用体和枚举类型

共用体（也称为联合）和枚举类型都是在 C 语言中为了让内容更加易读而引入的。实际上，只要有结构体和基本的数据类型就足够了，所以在汇编中，这些多余的东西都消失不见了。为了试试，写一段有共用体和枚举类型的代码，然后反汇编一下看看效果。

```
// 定义一个枚举类型
typedef enum {
    ENUM_1 = 1,
    ENUM_2 = 2,
    ENUM_3,
    ENUM_4,
} myenum;

// 定义一个结构体
typedef struct {
    int a;
    int b;
    int c;
} mystruct;

typedef union {
    mystruct s;
    myenum e[3];
} myunion;

int myfunction(int a,int b)
{
    unsigned char buf[100] = { 0 };
    myunion *uns = (myunion *)buf;
    int i;
    // 访问共用体，结构体，使用枚举类型的变量
    for(i=0;i<5;i++)
    {
        uns[i].s.a = 0;
```

```
        uns[i].s.b = 1;
        uns[i].e[2] = ENUM_4;
    }
    return 0;
}
```

反汇编的结果和 2.3 节基本没有区别。

```
for(i=0;i<5;i++)
00411A57  mov     dword ptr [i],0
00411A5E  jmp     myfunction+49h (411A69h)
00411A60  mov     eax,dword ptr [i]
00411A63  add     eax,1
00411A66  mov     dword ptr [i],eax
00411A69  cmp     dword ptr [i],5
00411A6D  jge     myfunction+83h (411AA3h)
{
    uns[i].s.a = 0;
00411A6F  mov     eax,dword ptr [i]     ;基本没有区别，直接
                                        ;去找.a 的地址
00411A72  imul    eax,eax,0Ch
00411A75  mov     ecx,dword ptr [uns]
00411A78  mov     dword ptr [ecx+eax],0
    uns[i].s.b = 1;
00411A7F  mov     eax,dword ptr [i]
00411A82  imul    eax,eax,0Ch
00411A85  mov     ecx,dword ptr [uns]
00411A88  mov     dword ptr [ecx+eax+4],1
    uns[i].e[2] = ENUM_4;
00411A90  mov     eax,dword ptr [i]
00411A93  imul    eax,eax,0Ch
00411A96  mov     ecx,dword ptr [uns]
00411A99  mov     dword ptr [ecx+eax+8],4
}
00411AA1  jmp     myfunction+40h (411A60h)
```

C 语言中一切花哨的东西在汇编里都消失不见了，和单纯用结构体没有区别。用枚举类型和一般常数也没有任何区别，因为共用体和枚举类型是为了方便程序员阅读而设计的，对于编译器处理的结果来说，它们并不引入额外的指令。

第3章　练习反汇编C语言程序

3.1　算法的反汇编

C 语言的各种控制流程在反汇编中绝对是最让人舒服的部分，而最痛苦的莫过于算法的部分。一个简单算法的反汇编代码就会复杂到让人眼花缭乱的程度，更不用说复杂的算法了。

3.1.1　算法反汇编代码分析

没有简单的办法来解决这个问题，只能靠读者的耐心和体力了。下面举一个 3×3 矩阵相乘的例子。

```
int myfunction(int a[3][3],int b[3][3],int c[3][3])
{
    int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
            c[i][j] = a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j];
    }
    return 0;
}
```

这段代码很简单易懂，不过，汇编的写法真的很富有挑战性：

```
int i,j;
for(i=0;i<3;i++)
00411A3E  mov     dword ptr [i],0
00411A45  jmp     myfunction+30h (411A50h)
00411A47  mov     eax,dword ptr [i]
00411A4A  add     eax,1
00411A4D  mov     dword ptr [i],eax
00411A50  cmp     dword ptr [i],3
00411A54  jge     myfunction+0AEh (411ACEh)
{
    for(j=0;j<3;j++)
00411A56  mov     dword ptr [j],0
00411A5D  jmp     myfunction+48h (411A68h)
00411A5F  mov     eax,dword ptr [j]
00411A62  add     eax,1
```

```
00411A65  mov     dword ptr [j],eax
00411A68  cmp     dword ptr [j],3
00411A6C  jge     myfunction+0A9h (411AC9h)
    c[i][j] = a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j];

;从这里开始的指令只有 mov,add 和 imul
;尝试把它们还原成表达式吧
00411A6E  mov     eax,dword ptr [i]
00411A71  imul    eax,eax,0Ch
00411A74  mov     ecx,dword ptr [a]
00411A77  mov     edx,dword ptr [j]
00411A7A  mov     esi,dword ptr [b]
00411A7D  mov     eax,dword ptr [ecx+eax]
00411A80  imul    eax,dword ptr [esi+edx*4]
00411A84  mov     ecx,dword ptr [i]
00411A87  imul    ecx,ecx,0Ch
00411A8A  mov     edx,dword ptr [a]
00411A8D  mov     esi,dword ptr [j]
00411A90  mov     edi,dword ptr [b]
00411A93  mov     ecx,dword ptr [edx+ecx+4]
00411A97  imul    ecx,dword ptr [edi+esi*4+0Ch]
00411A9C  add     eax,ecx
00411A9E  mov     edx,dword ptr [i]
00411AA1  imul    edx,edx,0Ch
00411AA4  mov     ecx,dword ptr [a]
00411AA7  mov     esi,dword ptr [j]
00411AAA  mov     edi,dword ptr [b]
00411AAD  mov     edx,dword ptr [ecx+edx+8]
00411AB1  imul    edx,dword ptr [edi+esi*4+18h]
00411AB6  add     eax,edx
00411AB8  mov     ecx,dword ptr [i]
00411ABB  imul    ecx,ecx,0Ch
00411ABE  add     ecx,dword ptr [c]
00411AC1  mov     edx,dword ptr [j]
00411AC4  mov     dword ptr [ecx+edx*4],eax
00411AC7  jmp     myfunction+3Fh (411A5Fh)
}
00411AC9  jmp     myfunction+27h (411A47h)
```

3.1.2 算法反汇编阅读技巧

要阅读上面那样的代码，首先把流程控制的代码与数值计算的代码分开是关键，因此前面的练习能起很大的帮助作用。得到数值计算的代码部分后，必须判断输入与输出

(一般自然是被读的内部变量为输入，被写的内部变量为输出)，然后把它还原成一个 C 语言的表达式。任何一段中间不加任何跳转，连续的 mov 和加减乘除的指令一般都可以还原为一个 C 表达式。当然，这可不是一个轻松的工作。

在这里顺便可以看到，二维数组 a[x][y]，处理等同于一个大小为 a[y]的结构长度为 x 的数组。所以，前面讲到的数组访问的代码非常有价值。基本的方法如下：

```
mov     eax,<我要取的数组元素的下标>
imul    eax,eax,<结构的大小>
mov     ecx,<结构数组开始的地址>
mov     eax,dword ptr [ecx+eax]        ;取得数组元素的内容
                                       ;放到 eax 中
```

访问结构内部变量的时候，最后面的一个指令还会加上一个数字：

```
mov     eax,dword ptr[ecx+eax+0Ch]
```

看到这样的代码，我们应该联想到表达式中含有的数组或结构体。在 3.3 节“汇编反 C 语言练习”中，我们将做一个类似的练习。

3.2　发行版的反汇编

基础知识

前面费了很多工夫解析代码，但那都是调试版本的代码。使用调试版本代码是入门的需要，因为这样汇编代码会更好理解。到非调试版本的时候，编译器将进行非常多的优化，使汇编代码变得精简的同时，阅读的困难也大大增加了，而且读者将碰到的代码显然都是发行版本。

举个函数调用过程的例子如下：

```
0040108D  push    eax
0040108E  push    esi
0040108F  call    myfunction1 (401000h)
00401094  add     esp,8
```

优化之后，两个参数根本没有放入内部变量中，而是放在 eax 和 esi 中直接传给函数了。然后调用 myfunction1，最后是恢复 esp，与调试版本的情况相同。

接下来是函数的执行过程，在堆栈中划分内部区域的代码可能被省略，因为内部变量在比较少的情况下，都直接使用寄存器了。把内部变量初始化为 int 3 的代码也被省略。取参数也不会放入内部变量中，而是用 esp 直接取。同时，有时 ebp 根本没被使用。

代码分析

下面是一个循环的简单函数。

```
int myfunction1(int a,int b)
{
00401000  push    ebx ;保存 ebx,esi,edi
00401001  push    esi
int i;
for(i=0;i<5;i++)

;取第一个参数。本来第一个参数是 esp+4h，但是因为前面有两
;个 push，所以变成 esp+0Ch
00401002  mov     esi,dword ptr [esp+0Ch]
00401006  push    edi

;取第二个参数，计算方法同上
00401007  mov     edi,dword ptr [esp+14h]

;清空 ebx，显然没有定义内部变量 i，而是直接用了 ebx
0040100B  xor     ebx,ebx
0040100D  lea     ecx,[ecx]                    ;无意义的指令
{
```

循环的结尾处是这样的：

```
00401020  inc     ebx
00401021  cmp     ebx,5
00401024  jl      myfunction1+10h (401010h)
```

可见优化后的 for 循环喜欢模仿相对简单的 do 循环的方式，把判断和跳转放在最后。

前面的矩阵相乘的例子如下，请注意这时候 C 代码和后面的反汇编代码已经没有一一对应的关系了。

```
int myfunction(int a[3][3],int b[3][3],int c[3][3])
{
    int i,j;
    for(i=0;i<3;i++)
00401000  mov     eax,dword ptr [esp+4]        ;a 保存到 eax 中
```

```
00401004  mov     edx,dword ptr [esp+0Ch]  ;c保存到edx中
00401008  mov     ecx,dword ptr [esp+8]    ;b保存到ecx中
0040100C  push    ebx
0040100D  push    esi

;这里已经开始处理数据。必须把这些指令和函数入口指令与
;循环控制指令分开
0040100E  add     eax,4
00401011  push    edi
00401012  add     edx,8    ;同前面的add eax,4
00401015  mov     esi,3                    ;循环变量取3
0040101A  lea     ebx,[ebx]                ;无意义指令
{
    for(j=0;j<3;j++)
        c[i][j] = a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j];

;显然从这里开始，是数值计算过程。代码的优化，使内部循环不见
;了。可以数一下imul指令，刚好9次，说明这里用一个单循环
;完成了原来双循环的工作
00401020  mov     ebx,dword ptr [eax]
00401022  imul    ebx,dword ptr [ecx+0Ch]
00401026  mov     edi,dword ptr [ecx+18h]
00401029  imul    edi,dword ptr [eax+4]
0040102D  add     edi,ebx
0040102F  mov     ebx,dword ptr [eax-4]
00401032  imul    ebx,dword ptr [ecx]
00401035  add     edi,ebx
00401037  mov     dword ptr [edx-8],edi
0040103A  mov     ebx,dword ptr [eax]
0040103C  imul    ebx,dword ptr [ecx+10h]
00401040  mov     edi,dword ptr [ecx+1Ch]
00401043  imul    edi,dword ptr [eax+4]
00401047  add     edi,ebx
00401049  mov     ebx,dword ptr [eax-4]
0040104C  imul    ebx,dword ptr [ecx+4]
00401050  add     edi,ebx
00401052  mov     dword ptr [edx-4],edi
00401055  mov     ebx,dword ptr [eax+4]
00401058  imul    ebx,dword ptr [ecx+20h]
0040105C  mov     edi,dword ptr [ecx+14h]
0040105F  imul    edi,dword ptr [eax]
00401062  add     edi,ebx
00401064  mov     ebx,dword ptr [eax-4]
00401067  imul    ebx,dword ptr [ecx+8]
```

```
0040106B  add     edi,ebx
0040106D  mov     dword ptr [edx],edi
0040106F  add     eax,0Ch
00401072  add     edx,0Ch
00401075  dec     esi                 ;这里开始两条是循环指令
00401076  jne     myfunction+20h (401020h)
00401078  pop     edi
00401079  pop     esi
    }
return 0;
0040107A  xor     eax,eax
0040107C  pop     ebx
}
```

在 3.3 节的练习中，读者将会看到将发行版本的一段反汇编代码，还原成 C 语言的过程。这里需要提醒读者的是，并非所有的反汇编代码，都能正确地还原成编译之前的 C 代码。虽然，写出有同样功能代码的 C 语言也许总是可能的，但是，试图完全还原出编译前的 C 语言，在很多情况下却是完全不可能的。

但是这并不重要。在这一章里，反汇编代码常常被反写成 C 语言，这是为了便于以 C 语言为“母语”的读者的理解与练习的需要。而在实际的需求中，最重要的是对代码逻辑的理解。无论是汇编还是 C 语言，只要能够理解，又何必拘泥于语言的形式呢？

关于破解（Crack）

对于商业软件的开发者而言，注册机、破解补丁真是挥之不去的噩梦。

许多软件通过让用户输入一个很长的注册码来确认用户的合法性，这个注册码是以某种方法计算出来的，那么它应当可以通过某种验算的考验。如果验证完全是在本地进行的（许多软件并不要求网络验证），一个容易想到的结果就是，这段验证的代码一定在本地的代码中。

那么找到并反汇编，再理解这段代码就成为破解者的主要爱好了。成功者可以编写出注册机，注册机总是产生一个能通过验算的注册码。当然注册机并不一定总是可以开发出来：这取决于是否能根据验算的方法逆向推导出生成注册码的方法。并非所有的算法都是可逆的，因此这并不一定总是可行的。

但是破解者马上就能找到更好的方法：直接修改二进制的验证程序，让它不分青红皂白地返回总是正确就 OK 了！破解补丁就是用来修改二进制验证代码的小程序，

或者直接就是已经修改过的软件主程序或验证组件。

对于破解的对抗，一些验证过程被设计为必须联网（比如 Windows 需要激活），这样避免验证算法放在本地泄露，但是这依然不能杜绝破解补丁，一些开发者设法让机器码更难以读懂（见本书的第 16～17 章）。这依然是一个永远不会结束的，矛与盾对抗的战场。

3.3　汇编反 C 语言练习

思考与练习

下面是一个练习，这个练习完全是发行版本的一个函数。如果能顺利完成这个练习，那么基本 C 语法的反汇编结果阅读也就入门了。

我们不公布 C 语言的源程序，但尝试把它还原成 C 语言。这次用的完全发行版本，经过 o2 优化的代码，足够接近实战了。下面的汇编语言程序对应一个完整的 C 函数。如果感觉困难，建议在看完后面的解题过程之后，再自己做一次这个练习。每行指令后的F、D、C 标记，是笔者加上用以识别代码类型的，将在下面的解法中解释。

```
00401000  push   ecx                              F
00401001  mov    ecx,dword ptr [esp+10h]          D
00401005  mov    edx,dword ptr [esp+8]            D
00401009  push   ebx                              F
0040100A  mov    ebx,dword ptr [esp+18h]          D
0040100E  push   esi                              F
0040100F  mov    esi,dword ptr [esp+14h]          D
00401013  push   ebp                              F
00401014  xor    eax,eax                          C
00401016  push   edi                              F
00401017  jmp    myfunction+20h (401020h)         F
00401019  lea    esp,[esp]                        F
00401020  mov    edi,dword ptr [esi+8]            D
00401023  imul   edi,dword ptr [edx+eax*8+4]      D
00401028  mov    ebp,dword ptr [esi]              D
0040102A  imul   ebp,dword ptr [edx+eax*8]        D
0040102E  add    edi,ebp                          D
00401030  mov    dword ptr [ecx],edi              D
00401032  mov    edi,dword ptr [esi+0Ch]          D
```

```
00401035  imul   edi,dword ptr [edx+eax*8+4]              D
0040103A  mov    ebp,dword ptr [esi+4]                    D
0040103D  imul   ebp,dword ptr [edx+eax*8]                D
00401041  add    edi,ebp                                  D
00401043  mov    ebp,dword ptr [ecx]                      D
00401045  add    ebp,edi                                  D
00401047  add    ebx,ebp                                  D
00401049  mov    dword ptr [ecx+4],edi                    D
0040104C  inc    eax                                      C
0040104D  add    ecx,8                                    D
00401050  cmp    eax,2                                    C
00401053  jl     myfunction+20h (401020h)                 C
00401055  call   rand (401138h)                           C
0040105A  add    ebx,eax                                  D
0040105C  cmp    ebx,64h                                  C
0040105F  pop    edi                                      F
00401060  pop    ebp                                      F
00401061  je     myfunction+7Bh (40107Bh)                 C
00401063  cmp    ebx,6Eh                                  C
00401066  je     myfunction+88h (401088h)                 C
00401068  push   offset string "nothing" (407114h)        F
0040106D  call   printf (401107h)                         F
00401072  add    esp,4                                    F
00401075  pop    esi                                      F
00401076  mov    eax,ebx                                  F
00401078  pop    ebx                                      F
00401079  pop    ecx                                      F
0040107A  ret                                             F
0040107B  push   offset string "cnt is 100" (407108h)     F
00401080  call   printf (401107h)                         F
00401085  add    esp,4                                    F
00401088  push   offset string "cnt is 110" (4070FCh)     F
0040108D  call   printf (401107h)                         F
00401092  add    esp,4                                    F
00401095  pop    esi                                      F
00401096  mov    eax,ebx                                  F
00401098  pop    ebx                                      F
00401099  pop    ecx                                      F
0040109A  ret                                             F
```

技术细节

本练习的解法大致如下。

第一步，把指令分成几类，并实际地把它们区分开，各个击破：首先是函数调用相关代码，这些代码用于调用函数或者作为一个函数被调用。几乎凡是堆栈操作（备份寄存器或者压入参数）可全部归入此类。此外还有 call 指令、堆栈恢复，这类代码很容易识别，把它们标记为 F，称之为 F 指令。

然后是流程控制代码，涉及判断和跳转指令，以及对循环变量操作的指令。这些应用于循环、判断语句，比较容易看出，标记为 C，称之为 C 指令。

剩余的是数据处理，应该不含有函数调用，多半不含有堆栈操作，也不会含有跳转（跳转已经归入流程控制中）。对于复杂的数据处理代码，只能逐行翻译，然后多行组合，标记为 D，我们称之为 D 指令。

标记已经做好了，请回头看上面的代码。当然，你可能出于不同的观点，和我做上的标记不同，或者同样的指令归类不同，这都没有关系。

第二步，取出其中标记为 D 的代码进行逐行翻译。首先标记为 F 的语句基本不需要翻译，它们本身就是简单的函数调用。其次标记为 C 的指令，将它们翻译为 if、for、do 或者 switch，工作相对简单。

下面的指令取自上面的前半段的 D 指令，我删除了其中夹杂的 F 指令。而 C 指令则是容易翻译的部分，那么我们尝试改写 D 指令为如下最右边的表达式，其中我用了 p1、p2、p3 表示函数的参数 1、参数 2、参数 3。取参数时针对 esp 的偏移，随着 push 指令的执行而有所变动，切不可机械计算。

```
00401001  mov    ecx,dword ptr [esp+10h]       D   ecx = p3
00401005  mov    edx,dword ptr [esp+8]        D   edx = p1
0040100A  mov    ebx,dword ptr [esp+18h]      D   ebx = p4
0040100F  mov    esi,dword ptr [esp+14h]      D   esi = p2
00401014  xor    eax,eax                      C
00401020  mov    edi,dword ptr [esi+8]        D   edi = p2[2]
00401023  imul   edi,dword ptr [edx+eax*8+4]  D   edi*=p1[eax*2+1]
00401028  mov    ebp,dword ptr [esi]          D   ebp = p2[0]
0040102A  imul   ebp,dword ptr [edx+eax*8]    D   ebp *= p1[eax*2]
0040102E  add    edi,ebp                      D   edi += ebp
00401030  mov    dword ptr [ecx],edi          D   ecx [0] = edi
00401032  mov    edi,dword ptr [esi+0Ch]      D   edi = p2[3]
```

```
00401035  imul    edi,dword ptr [edx+eax*8+4]  D   edi *= p1[eax*2+1]
0040103A  mov     ebp,dword ptr [esi+4]       D   ebp = p2[1]
0040103D  imul    ebp,dword ptr [edx+eax*8]   D   ebp *= p1[eax*2]
00401041  add     edi,ebp                     D   edi += ebp
00401043  mov     ebp,dword ptr [ecx]         D   ebp = ecx [0]
00401045  add     ebp,edi                     D   ebp += edi
00401047  add     ebx,ebp                     D   ebx += ebp
00401049  mov     dword ptr [ecx+4],edi       D   ecx [1] = edi
0040104C  inc     eax                         C
0040104D  add     ecx,8                       D   ecx+=8
00401050  cmp     eax,2                       C
00401053  jl      myfunction+20h (401020h)    C
```

第三步，自然是表达式的合并与控制流程的结合了，前面 D 系列的表达式，把中间过程除去，已经很容易得到表达式如下，其中 eax 开始为 0，ecx 开始为 p3。

```
ecx [0] = p2[2]*p1[eax*2+1]+p2[0]*p1[eax*2];
ecx [1] = p2[3]*p1[eax*2+1]+p2[1]*p1[eax*2];
```

此外，jl 明显导致了一个循环，每次循环的更改是：

```
eax++;
ecx+=8;
```

eax 是循环变量，ecx 显然用于取数组位置，且二者保持固定的同步增加，可以用一个循环变量替代，所以翻译结果如下：

```
for(i=0;i<2;i++)
{
    p3[i] = p2[2]*p1[2*i+1]+p2[0]*p1[2*i];
    p3[i+1] = p2[3]*p1[2*i+1]+p2[1]*p1[i*2];
}
```

好了，以上就是三步的解法。后面的部分，也请如此完成，值得注意的是以下部分：

```
0040105C  cmp     ebx,64h                     C
0040105F  pop     edi                         F
00401060  pop     ebp                         F
00401061  je      myfunction+7Bh (40107Bh)     C
00401063  cmp     ebx,6Eh                     C
00401066  je      myfunction+88h (401088h)    C
```

除去其中的 F 指令，连续看到 cmp 和 je，说明这是一个 switch。这和调试版本没有什么区别，翻译的难度也非常小，这里就不再重述详细的过程了。现在请读者写出完整的答案吧。

基础篇

内核编程

本书的第二部分，是编写 Windows 内核程序编程方法的基础。本部分包括第 4~7 章，如果读者对 Windows 内核编程已经有一定的了解，可以跳过本部分；如果读者从未接触过 Windows 内核编程，本部分将指导读者开始 Windows 内核编程，学会使用 WDK，并熟悉内核编程的习惯与方法。

第 4 章　内核字符串与内存

从本章开始讲解 Windows 内核编程的基础知识。在这里，Windows 内核编程与 Windows 驱动编程被认为是完全等同的概念，不做区分。所以下文的“驱动编程”和“内核编程”是同一个意义。

本书的目标是理解 Windows 内核。对于普通的 Windows 应用软件的开发人员来说，主要的障碍有两个：一是 Windows 没有源码提供，所以有时不得不看汇编语言；二是对内核的 C 语言编程方法不熟悉。

所以从这一章开始读者可以学习用 C 语言开发 Windows 内核程序的编程方法，也就是 Windows 驱动程序。请注意，本章讲述如何用 WDK 编写驱动程序，但是，目的是为了让读者可以读懂用 C 语言编写的内核程序。所以这里没有讲述 WDK 开发，以及调试环境的配置。读者可以在第三部分“探索篇 研究内核”的开始部分，学到配置开发环境，编译一个驱动程序的方法。在第四部分将引导读者开发一个实例，会用到这些编程方法。

因为以下的代码编译、运行与调试都需要安装 WDK，并包含 WDK 相应头文件，并使用 WinDbg 进行调试。参阅第三部分“探索篇 研究内核”的开始部分（第 8 章“进入 Windows 内核”）之前，读者可以先只学习编程方法。

4.1　字符串的处理

4.1.1　使用字符串结构

常常使用传统 C 语言的程序员比较喜欢用如下的方法定义和使用字符串：

```
char*str = { "my first string" };              // Ansi 字符串
wchar_t *wstr = { L"my first string" };        // Unicode 字符串
size_t len = strlen(str);                      // Ansi 字符串求长度
size_t wlen = wcslen(wstr);                    // Unicode 字符串求长度
printf("%s %ws %d %d",str,wstr,len,wlen);      // 打印两种字符串
```

但是实际上这种字符串相当的不安全，很容易导致缓冲溢出漏洞。这是因为没有任何地方确切地表明一个字符串的长度，仅仅用一个“\0”字符来标明这个字符串的结束。一旦碰到根本就没有空结束的字符串（可能是攻击者恶意的输入，或者是编程错误导致的意外），程序就可能陷入崩溃。

使用高级 C++特性的编码者则容易忽略这个问题，因为常常使用 std::string 和 CString 这样高级的类，不用去担忧字符串的安全性。

在驱动开发中，一般不再用空来表示一个字符串的结束，而是定义了如下的一个结构：

```
typedef struct _UNICODE_STRING {
    USHORT Length;                  // 字符串的长度（字节数）
    USHORT MaximumLength;           // 字符串缓冲区的长度（字节数）
    PWSTR  Buffer;                  // 字符串缓冲区
} UNICODE_STRING, *PUNICODE_STRING;
```

以上是 Unicode 字符串，一个字符为双字节。与之对应的还有一个 Ansi 字符串，Ansi 字符串就是 C 语言中常用的单字节表示一个字符的窄字符串。

```
typedef struct _STRING {
    USHORT Length;
    USHORT MaximumLength;
    PSTR Buffer;
} ANSI_STRING, *PANSI_STRING
```

在驱动开发中到处可见的是 Unicode 字符串，因此可以说：Windows 的内核是使用 Uincode 编码的。ANSI_STRING 仅仅在某些碰到窄字符的场合使用，而且这种场合非常罕见。

Windows 内核使用 Unicode 编码。内核中所见字符串大多为 UNICODE_STRING 结构。

UNICODE_STRING 并不保证 Buffer 中的字符串是以空结束的。因此，类似下面的做法都是错误的，可能会导致内核崩溃：

```
UNICODE_STRING str;
…
len = wcslen(str.Buffer);               // 试图求长度
DbgPrint("%ws",str.Buffer);             // 试图打印 str.Buffer
```

如果要用以上的方法，必须在编码中保证 Buffer 始终是以空结束，但这又是一个麻烦的问题。所以，使用微软提供的 Rtl 系列函数来操作字符串，才是正确的方法。下文逐步地讲述这个系列的函数的使用。

4.1.2 字符串的初始化

请回顾之前的 UNICODE_STRING 结构。读者应该可以注意到，这个结构中并不含有字符串缓冲的空间。这是一个初学者常见的出问题的来源。以下的代码是完全错误的，内核会立刻崩溃：

```
UNICODE_STRING str;
wcscpy(str.Buffer,L"my first string!");
str.Length = str.MaximumLength =
wcslen(L"my first string!") * sizeof(WCHAR);
```

以上的代码定义了一个字符串并试图初始化它的值。但是非常遗憾，这样做是不对的。因为 str.Buffer 只是一个未初始化的指针，它并没有指向有意义的空间。相反，以下的方法是正确的：

```
UNICODE_STRING str;
str.Buffer = L"my first string!";
str.Length = str.MaximumLength =
wcslen(L"my first string!") * sizeof(WCHAR);
```

实际上，明显的初始化写法如下：

```
//请分析一下为何这样写是对的
UNICODE_STRING str = {
    sizeof(L"my first string!") - sizeof((L"my first string!")[0]),
    sizeof(L"my first string!"),
    L"my first_string!" };
```

但是这样定义一个字符串实在太烦琐了，但是在头文件 ntdef.h 中有一个宏方便这种定义。使用这个宏之后，我们就可以简单地定义一个常数字符串如下：

```
#include <ntdef.h>
UNICODE_STRING str = RTL_CONSTANT_STRING(L"my first string!");
```

这只能在定义这个字符串的时候使用。为了随时初始化一个字符串，可以使用 RtlInitUnicodeString。示例如下：

```
UNICODE_STRING str;
RtlInitUnicodeString(&str,L"my first string!");
```

用本节的方法初始化的字符串，不用担心内存释放方面的问题，因为并没有分配任何内存。

4.1.3 字符串的拷贝

因为字符串不再是空结束的，所以使用 wcscpy 来拷贝字符串是不行的。UNICODE_STRING 可以用 RtlCopyUnicodeString 来进行拷贝，在进行这种拷贝的时候，最需要注意的一点是：拷贝目的字符串的 Buffer 必须有足够的空间。如果 Buffer 的空间不足，字符串会拷贝不完全。这是一个比较隐蔽的错误。

下面举一个例子：

```
UNICODE_STRING dst;         // 目标字符串
WCHAR dst_buf[256];         // 我们现在还不会分配内存，所以先定义缓冲区
UNICODE_STRING src = RTL_CONST_STRING(L"My source string!");

// 把目标字符串初始化为拥有缓冲区长度为 256 的 UNICODE_STRING 空串
RtlInitEmptyString(dst,dst_buf,256*sizeof(WCHAR));
RtlCopyUnicodeString(&dst,&src); // 字符串拷贝
```

以上这个拷贝之所以会成功，是因为 256 比 L"My source string! "的长度要大。如果小，则拷贝也不会出现任何明示的错误。但是拷贝结束之后，与使用者的目标不符，字符串实际上被截短了。

使用 UNICODE_STRING 结构的时候常常要自己确保其缓冲区的长度。

笔者曾经犯过的一个错误是没有调用 RtlInitEmptyString，结果 dst 字符串被初始化认为缓冲区长度为 0。虽然程序没有崩溃，却实际上没有拷贝任何内容。

在拷贝之前，最谨慎的方法是根据源字符串的长度动态分配空间。在 4.2 节“内存与链表”中，读者会看到动态分配内存处理字符串的方法。

4.1.4 字符串的连接

UNICODE_STRING 不再是简单的字符串，操作这个数据结构往往需要更多的耐心。读者会常常碰到这样的需求：要把两个字符串连接到一起。简单地追加一个字符串并不困难，重要的依然是保证目标字符串的空间大小。下面是范例：

```
NTSTATUS status;
UNICODE_STRING dst;          // 目标字符串
WCHAR dst_buf[256];          // 我们现在还不会分配内存，所以先定义缓冲区
UNICODE_STRING src = RTL_CONST_STRING(L"My source string!");
```

```
// 把目标字符串初始化为拥有缓冲区长度为 256 的 UNICODE_STRING 空串
RtlInitEmptyString(dst,dst_buf,256*sizeof(WCHAR));
RtlCopyUnicodeString(&dst,&src); // 字符串拷贝

status = RtlAppendUnicodeToString(
        &dst,L"my second string!");
if(status != STATUS_SUCCESS)
{
    … …
}
```

NTSTATUS 是常见的返回值类型，如果函数成功，返回 STATUS_SUCCESS；否则，是一个错误码。RtlAppendUnicodeToString 在目标字符串空间不足的时候依然可以连接字符串，但是会返回一个警告性的错误 STATUS_BUFFER_TOO_SMALL。

另外一种情况是希望连接两个 UNICODE_STRING，这种情况请调用 RtlAppendUnicodeStringToString。这个函数的第二个参数也是一个 UNICODE_STRING 的指针。

4.1.5　字符串的打印

字符串连接的另一种常见的情况是字符串和数字的组合。有时数字需要被转换为字符串，有时需要把若干个数字和字符串混合组合起来。这往往用于打印日志的时候，日志中可能含有文件名、时间和行号，以及其他的信息。

熟悉 C 语言的读者会使用 sprintf，这个函数的宽字符版本为 swprintf。该函数在驱动开发中依然可以使用，但是不安全，微软建议使用 RtlStringCbPrintfW 来代替它。RtlStringCbPrintfW 需要包含头文件 ntstrsafe.h，在连接的时候，还需要连接库 ntsafestr.lib。

下面的代码生成一个字符串，字符串中包含文件的路径和这个文件的大小。

```
#include <ntstrsafe.h>
// 任何时候，假设文件路径的长度为有限的都是不对的。应该动态地分配
// 内存。但是动态分配内存的方法还没有讲述，所以这里再次把内存空间
// 定义在局部变量中，也就是所谓的“在栈中”
WCHAR buf[512] = { 0 };
UNICODE_STRING dst;
NTSTATUS status;
……
```

```
// 字符串初始化为空串。缓冲区长度为 512*sizeof(WCHAR)
RtlInitEmptyString(dst,dst_buf,512*sizeof(WCHAR));

// 调用 RtlStringCbPrintfW 来进行打印
status = RtlStringCbPrintfW(
    dst->Buffer,512*sizeof(WCHAR),L"file path = %wZ file size = %d \r\n",
    &file_path,file_size);
// 这里调用 wcslen 没问题，这是因为 RtlStringCbPrintfW 打印的
// 字符串是以空结束的
dst->Length = wcslen(dst->Buffer) * sizeof(WCHAR);
```

RtlStringCbPrintfW 在目标缓冲区内存不足的时候依然可以打印，但是多余的部分被截去了，返回的 status 值为 STATUS_BUFFER_OVERFLOW。调用这个函数之前很难知道究竟需要多长的缓冲区，一般都采取倍增尝试。每次都传入一个为前次尝试长度2倍的新缓冲区，直到这个函数返回 STATUS_SUCCESS 为止。

值得注意的是，UNICODE_STRING 类型的指针，用%wZ 打印可以打印出字符串，在不能保证字符串为空结束的时候，必须避免使用%ws 或者%s。其他的打印格式字符串与传统 C 语言中的 printf 函数完全相同，可以尽情使用。

另外就是常见的输出打印。printf 函数只有在有控制台输出的情况下才有意义，在驱动中没有控制台，但是 Windows 内核中拥有调试信息输出机制，可以使用 WinDbg 查看打印的调试信息。

驱动中可以调用 DbgPrint()函数来打印调试信息。这个函数的使用和 printf 基本相同，但是格式字符串要使用宽字符。DbgPrint()的一个缺点在于，发行版本的驱动程序往往不希望附带任何输出信息，只有调试版本才需要调试信息。但是 DbgPrint()无论是发行版本还是调试版本编译都会有效，为此可以自己定义一个宏：

```
#if DBG
    KdPrint(a)   DbgPrint##a
#else
    KdPrint (a)
#endif
```

不过这样做的后果是，由于 KdPrint (a)只支持 1 个参数，因此必须把 DbgPrint 的所有参数都括起来当做一个参数传入。导致 KdPrint 看起来很奇特的是用了双重括弧：

```
// 调用 KdPrint 来进行输出调试信息
status = KdPrint ((
    L"file path = %wZ file size = %d \r\n",
```

```
    &file_path,file_size));
```

这个宏没有必要自己定义，WDK 包中已有，所以可以直接使用 KdPrint 来代替 DbgPrint 取得更方便的效果。

4.2　内存与链表

4.2.1　内存的分配与释放

内存泄漏是 C 语言中一个臭名昭著的问题。但是作为内核开发者，读者将有必要自己来面对它。在传统的 C 语言中，分配内存常常使用的函数是 malloc。这个函数的使用非常简单，传入长度参数就得到内存空间。在驱动中使用内存分配，这个函数不再有效。驱动中分配内存，最常用的是调用 ExAllocatePoolWithTag，其他的方法在本章范围内全部忽略。回忆前一节关于字符串处理的情况。一个字符串被复制到另一个字符串的时候，最好根据源字符串的空间长度来分配目标字符串的长度。下面的例子，是把一个字符串 src 拷贝到字符串 dst。

```
// 定义一个内存分配标记
#define MEM_TAG  'MyTt'
// 目标字符串，接下来它需要分配空间
UNICODE_STRING dst = { 0 };
// 分配空间给目标字符串。根据源字符串的长度
dst.Buffer =
 (PWCHAR)ExAllocatePoolWithTag(NonpagedPool,src->Length,MEM_TAG);
if(dst.Buffer == NULL)
{
    // 错误处理
    status = STATUS_INSUFFICIENT_RESOUCRES;
    ……
}
dst.Length = dst.MaximumLength = src->Length;
status = RtlCopyUnicodeString(&dst,&src);
ASSERT(status == STATUS_SUCCESS);
```

ExAllocatePoolWithTag 的第一个参数 NonpagedPool 表明分配的内存是锁定内存。这些内存永远真实存在于物理内存上，不会被分页交换到硬盘上去。第二个参数是长度。第三个参数是一个所谓的“内存分配标记”。

内存分配标记用于检测内存泄漏。想象一下，我们根据占用越来越多的内存的分配标记，就能大概知道泄漏的来源。一般每个驱动程序定义一个自己的内存标记，也可以在每个模块中定义单独的内存标记。内存标记是随意的 32 位数字，即使冲突也不会有什么问题。

内核中最常见的分配内存的方法是调用 ExAllocatePoolWithTag。

此外也可以分配可分页内存，使用 PagedPool 即可。

ExAllocatePoolWithTag 分配的内存可以使用 ExFreePool 来释放，如果不释放，则永远泄漏，并不像用户进程关闭后自动释放所有分配的空间。即使驱动程序动态卸载，也不能释放空间，唯一的办法是重启计算机。

ExFreePool 只需要提供所释放的指针即可。举例如下：

```
ExFreePool(dst.Buffer);
dst.Buffer = NULL;
dst.Length = dst.MaximumLength = 0;
```

ExFreePool 不能用来释放一个栈空间的指针，否则系统立刻崩溃。像以下的代码：

```
UNICODE_STRING src = RTL_CONST_STRING(L"My source string!");
ExFreePool(src.Buffer);
```

会导致立刻蓝屏，所以请务必保持 ExAllocatePoolWithTag 和 ExFreePool 的成对关系。

4.2.2 使用 LIST_ENTRY

Windows 的内核开发者们自己开发了部分数据结构，比如 LIST_ENTRY。

LIST_ENTRY 是一个双向链表结构，它总是在使用的时候被插入到已有的数据结构中。下面举一个例子。构筑一个链表，这个链表的每个节点，是一个文件名和一个文件大小两个数据成员组成的结构。此外有一个 FILE_OBJECT 的指针对象，在驱动中这代表一个文件对象（本书后面的章节会详细解释）。这个链表的作用是：保存了文件的文件名和长度。只要传入 FILE_OBJECT 的指针，使用者就可以遍历链表找到文件名和文件长度。

```
typedef struct {
    PFILE_OBJECT file_object;
    UNICODE_STRING file_name;
    LARGE_INTEGER file_length;
```

```
} MY_FILE_INFOR, *PMY_FILE_INFOR;
```

一些读者会马上注意到文件的长度用 LARGE_INTEGER 表示，这是一个代表长长整型的数据结构。这个结构我们在下一节“使用长长整型数据”中介绍。

Windows 内核中使用 LIST_ENTRY 作为链表，这个结构几乎随处可见。

为了让上面的结构成为链表节点，必须在里面插入一个 LIST_ENTRY 结构。至于插入的位置无所谓，可以放在最前，也可以放在中间，或者最后面。但是实际上读者很快会发现把 LIST_ENTRY 放在开头是最简单的做法：

```
typedef struct {
    LIST_ENTRY list_entry;
    PFILE_OBJECT file_object;
    UNICODE_STRING file_name;
    LARGE_INTEGER file_length;
} MY_FILE_INFOR, *PMY_FILE_INFOR;
```

LIST_ENTRY 如果是作为链表的头，在使用之前，必须调用 InitializeListHead 来初始化。下面是示例的代码：

```
// 我们的链表头
LIST_ENTRY        my_list_head;

// 链表头初始化。一般地应该在程序入口处调用一下
void MyFileInforInilt()
{
    InitializeListHead(&my_list_head);
}

// 我们的链表节点，里面保存一个文件名和一个文件长度信息
typedef struct {
    LIST_ENTRY list_entry;
    PFILE_OBJECT file_object;
    PUNICODE_STRING file_name;
    LARGE_INTEGER file_length;
} MY_FILE_INFOR, *PMY_FILE_INFOR;

// 追加一条信息，也就是增加一个链表节点。请注意 file_name 是外面分配的
// 内存由使用者管理，本链表并不管理它
NTSTATUS MyFileInforAppendNode(
    PFILE_OBJECT file_object,
    PUNICODE_STRING file_name,
    PLARGE_INTEGER file_length)
```

```
{
    PMY_FILE_INFOR my_file_infor =
        (PMY_FILE_INFOR)ExAllocatePoolWithTag(
            PagedPool,sizeof(MY_FILE_INFOR),MEM_TAG);
    if(my_file_infor == NULL)
        return STATUS_INSUFFICIENT_RESOURES;

    // 填写数据成员
    my_file_infor->file_object = file_object;
    my_file_infor->file_name = file_name;
    my_file_infor->file_length = file_length;

    // 插入到链表末尾。请注意这里没有使用任何锁。所以，这个函数不是
    // 多线程安全的。在下面自旋锁的使用中讲解如何保证多线程安全性
    InsertHeadList(&my_list_head, (PLIST_ENTRY)& my_file_infor);
        return STATUS_SUCCESS;
}
```

以上的代码实现了插入。可以看到 LIST_ENTRY 插入到 MY_FILE_INFOR 结构的头部的好处。这样一来，一个 MY_FILE_INFOR 看起来就像一个 LIST_ENTRY。不过糟糕的是，并非所有的情况都可以这样。比如 MS 的许多结构喜欢一开头就是结构的长度，因此在通过 LIST_ENTRY 结构的地址获取所在的节点的地址时，有个地址偏移计算的过程。可以在下面的一个典型的遍历链表的示例中看到：

```
for(p = my_list_head.Flink; p != &my_list_head.Flink; p = p->Flink)
{
    PMY_FILE_INFOR elem =
        CONTAINING_RECORD(p,MY_FILE_INFOR, list_entry);
    // To do something here…
}
```

其中的 CONTAINING_RECORD 是一个 WDK 中已经定义的宏，作用是通过一个 LIST_ENTRY 结构的指针，找到这个结构所在的节点的指针。定义如下：

```
#define CONTAINING_RECORD(address, type, field) ((type *)( \
    (PCHAR)(address) - \
    (ULONG_PTR)(&((type *)0)->field)))
```

从上面的代码中可以总结如下：

LIST_ENTRY 中的数据成员 Flink 指向下一个 LIST_ENTRY。整个链表中的最后一个 LIST_ENTRY 的 Flink 不是空，而是指向头节点。得到 LIST_ENTRY 之后，要用 CONTAINING_RECORD 来得到链表节点中的数据。

4.2.3 使用长长整型数据

这里解释前面碰到的 LARGE_INTEGER 结构。与可能的误解不同，64 位数据并非要在 64 位操作系统下才能使用。在 VC 中，64 位数据的类型为__int64。定义写法如下：

```
__int64 file_offset;
```

上面之所以定义的变量名为 file_offset，是因为文件中的偏移量是一种常见的要使用 64 位数据的情况。同时，文件的大小也是如此（回忆上一节中定义的文件大小）。32 位数据无符号整型只能表示到 4GB，而众所周知，现在超过 4GB 的文件绝对不罕见了。但是实际上__int64 这个类型在驱动开发中很少被使用，基本上被使用到的是一个共用体：LARGE_INTEGER。这个共用体定义如下：

```
typedef __int64 LONGLONG;
typedef union _LARGE_INTEGER {
    struct {
        ULONG LowPart;
        LONG HighPart;
    };
    struct {
        ULONG LowPart;
        LONG HighPart;
    } u;
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

这个共用体的方便之处在于，既可以很方便地得到高 32 位、低 32 位，也可以方便地得到整个 64 位。进行运算和比较的时候，使用 QuadPart 即可。

```
LARGE_INTEGER a,b;
a.QuadPart = 100;
a.QuadPart *= 100;
b.QuadPart = a.QuadPart;
if(b.QuadPart > 1000)
{
    KdPrint("b.QuadPart < 1000, LowPart = %x HighPart = %x", b.LowPart,
b.HighPart);
}
```

Windows 内核中如果一个函数参数要使用 64 位数据，一般是 PLARGE_INTEGER 类型。

上面这段代码演示了这种结构的一般用法。在实际编程中，会碰到大量的参数是 PLARGE_INTEGER 类型的。

4.2.4 使用自旋锁

链表之类的结构总是涉及恼人的多线程同步问题，这时候就必须使用锁。这里只介绍最简单的自旋锁。

有些读者可能疑惑锁存在的意义。这和多线程操作有关。在驱动开发的代码中，大多是存在于多线程执行环境的，也就是说，可能有几个线程在同时调用当前函数。

这样一来，4.2.2 节中提及的追加链表节点函数就根本无法使用了，因为 MyFileInforAppendNode 这个函数只是简单地操作链表。如果两个线程同时调用这个函数来操作链表的话（注意：这个函数操作的是一个全局变量链表），换句话说，无论有多少个线程同时执行，它们操作的都是同一个链表。这就可能发生一个线程插入一个节点的同时，另一个线程也同时插入的情况。它们都插入同一个链表节点的后边，这时链表就会发生问题。到底最后插入的是哪一个呢？要么一个丢失了，要么链表被损坏了。

如下的代码初始化获取一个自旋锁：

```
KSPIN_LOCK my_spin_lock;
KeInitializeSpinLock(&my_spin_lock);
```

KeInitializeSpinLock 这个函数没有返回值。下面的代码展示了如何使用这个 SpinLock。在 KeAcquireSpinLock 和 KeReleaseSpinLock 之间的代码是只有单线程执行的，其他的线程会停留在 KeAcquireSpinLock 等候，直到 KeReleaseSpinLock 被调用。KIRQL 是一个中断级。KeAcquireSpinLock 会提高当前的中断级，但是目前忽略这个问题，中断级在后面讲述。

```
KIRQL irql;
KeAcquireSpinLock(&my_spin_lock,&irql);
// To do something …
KeReleaseSpinLock(&my_spin_lock,irql);
```

初学者要注意的是，像下面写的这样的“加锁”代码是没有意义的，等于没加锁：

```
void MySafeFunction()
{
    KSPIN_LOCK my_spin_lock;
    KIRQL irql;
    KeInitializeSpinLock(&my_spin_lock);
```

```
    KeAcquireSpinLock(&my_spin_lock,&irql);
    // To do something …
    KeReleaseSpinLock(&my_spin_lock,irql);
}
```

原因是 my_spin_lock 在堆栈中，每个线程来执行的时候都会重新初始化一个锁。只有所有的线程共用一个锁，锁才有意义。所以，锁一般不会定义成局部变量，可以使用静态变量、全局变量，或者分配在堆中（参见 4.2.1 节“内存的分配与释放”）。请读者自己写出正确的方法。

LIST_ENTRY 有一系列的操作，这些操作并不需要使用者自己调用获取与释放锁，只需要为每个链表定义并初始化一个锁即可：

```
LIST_ENTRY   my_list_head;    // 链表头
KSPIN_LOCK   my_list_lock;        // 链表的锁

// 链表初始化函数
void MyFileInforInilt()
{
    InitializeListHead(&my_list_head);
    KeInitializeSpinLock(&my_list_lock);
}
```

链表一旦完成了初始化，之后的就可以采用一系列加锁的操作来代替普通的操作。比如插入一个节点，普通的操作代码如下：

```
InsertHeadList(&my_list_head, (PLIST_ENTRY)& my_file_infor);
```

换成加锁的操作方式如下：

```
ExInterlockedInsertHeadList(
    &my_list_head,
    (PLIST_ENTRY)& my_file_infor,
    &my_list_lock);
```

注意不同之处在于，增加了一个 KSPIN_LOCK 的指针作为参数。在 ExInterlockedInsertHeadList 中，会自动使用这个 KSPIN_LOCK 进行加锁。类似的还有一个加锁的 Remove 函数，用来移除一个节点，调用如下：

```
my_file_infor = ExInterlockedRemoveHeadList (
    &my_list_head,
    &my_list_lock);
```

这个函数从链表中移除第一个节点，并返回到 my_file_infor 中。

第 5 章　文件与注册表操作

操作文件和注册表在 Windows 系统中非常重要。开发过 Windows 应用程序的朋友应该使用过许多 API 函数，如 CreateFile、ReadFile、WriteFile、CreateKey、SetKeyValue。在内核编程中不能直接调用 WindowsAPI 函数，但是这些 API 函数在底层都有对应的内核实现。实际上，类似 CreateFile 这样的 API 函数在进入 R0 层（特权等级在后面的第 12 章中解释）之后，都由对应的内核函数来实现。

5.1　文件操作

5.1.1　使用 OBJECT_ATTRIBUTES

一般的想法是，打开文件应该传入这个文件的路径。但是内核并不直接接受一个字符串，使用者必须首先填写一个 OBJECT_ATTRIBUTES 结构。在文档中并没有公开这个 OBJECT_ATTRIBUTES 结构，但这个结构总是被 InitializeObjectAttributes 初始化。

下面专门说明 InitializeObjectAttributes。

```
VOID InitializeObjectAttributes(
    OUT POBJECT_ATTRIBUTES InitializedAttributes,
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes,
    IN HANDLE RootDirectory,
    IN PSECURITY_DESCRIPTOR SecurityDescriptor);
```

读者需要注意的是：InitializedAttributes 是要初始化的 OBJECT_ATTRIBUTES 结构的指针。ObjectName 则是对象名字字符串，也就是前文所描述的文件路径（如果要打开的对象是一个文件的话）。

在 Windows 内核中，无论是打开文件、打开注册表键还是打开设备，都会先调用初始化一个 OBJECT_ATTRIBUTES。

Attributes 则只需要填写 OBJ_CASE_INSENSITIVE| OBJ_KERNEL_HANDLE 即可（如果读者是想要方便、简洁地打开一个文件的话）。OBJ_CASE_INSENSITIVE 意味着名字字符串是不区分大小写的。由于 Windows 的文件系统本来就不区分字母大小写，所以笔者并没有尝试过如果不设置这个标记会有什么后果。OBJ_KERNEL_HANDLE 表明打开的文件句柄是一个“内核句柄”。内核文件句柄比应用层句柄使用更方便，可以

不受线程和进程的限制，在任何线程中都可以读写；同时打开内核文件句柄不需要顾及当前进程是否有权限访问该文件的问题（如果是有安全权限限制的文件系统）。如果不使用内核句柄，则有时不得不填写后面的 SecurityDescriptor 参数。

RootDirectory 用于相对打开的情况，目前省略，读者传入 NULL 即可。

SecurityDescriptor 用于设置安全描述符。由于笔者总是打开内核句柄，所以很少设置这个参数。

5.1.2 打开和关闭文件

下面的函数用于打开一个文件。

```
NTSTATUS ZwCreateFile(
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttribute,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PLARGE_INTEGER AllocationSize OPTIONAL,
    IN ULONG FileAttributes,
    IN ULONG ShareAccess,
    IN ULONG CreateDisposition,
    IN ULONG createOptions,
    IN PVOID EaBuffer OPTIONAL,
    IN ULONG EaLength);
```

这个函数的参数异常复杂，下面逐个说明一下。

FileHandle：一个句柄的指针。如果这个函数调用返回成功（STATUS_SUCCESS），那么打开的文件句柄就返回在这个地址内。

DesiredAccess：申请的权限。如果打开写文件内容，请使用 FILE_WRITE_DATA；如果需要读文件内容，请使用 FILE_READ_DATA；如果需要删除文件或者把文件改名，请使用 DELETE；如果想设置文件属性，请使用 FILE_WRITE_ATTRIBUTES。反之，读文件属性则使用 FILE_READ_ATTRIBUTES。这些条件可以用|（位或）来组合。有两个宏分别组合了常用的读权限和常用的写权限，分别为 GENERIC_READ 和 GENERIC_WRITE。还有一个宏代表全部权限，是 GENERIC_ALL。此外，如果想同步地打开文件，请加上 SYNCHRONIZE。同步打开文件详见后面对 CreateOptions 的说明。

ObjectAttribute：对象描述。见前一小节。

IoStatusBlock：也是一个结构。这个结构在内核开发中经常使用，它往往用于表示一个操作的结果。这个结构在文档中是公开的，如下：

```
typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status;
        PVOID Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

实际编程中很少用到 Pointer。一般地，返回的结果在 Status 中，成功则为 STATUS_SUCCESS；否则是一个错误码。进一步的信息在 Information 中，不同的情况下返回的 Information 的信息意义不同。针对 ZwCreateFile 调用的情况，Information 的返回值有以下几种可能：

- FILE_CREATED：文件被成功地新建了。
- FILE_OPENED：文件被打开了。
- FILE_OVERWRITTEN：文件被覆盖了。
- FILE_SUPERSEDED：文件被替代了。
- FILE_EXISTS：文件已存在（因而打开失败了）。
- FILE_DOES_NOT_EXIST：文件不存在（因而打开失败了）。

这些返回值和打开文件的意图有关（有时希望打开已存在的文件，有时则希望建立新的文件，等等）。这些意图在本节稍后的内容中会详细说明。

ZwCreateFile 的下一个参数是 AllocationSize。这个参数很少使用，请设置为 NULL。

再接下来的一个参数为 FileAttributes。这个参数控制新建立的文件的属性，一般地，设置为 FILE_ATTRIBUTE_NORMAL 即可。在实际编程中，笔者没有尝试过其他的值。

ShareAccess 是一个非常容易被人误解的参数。实际上，这是在本代码打开这个文件的时候，允许别的代码同时打开这个文件所持有的权限，所以称为共享访问。一共有 3 种共享标记可以设置：FILE_SHARE_READ、FILE_SHARE_WRITE 和 FILE_SHARE_DELETE。这 3 个标记可以用|（位或）来组合。举例如下：如果本次打开只使用了 FILE_SHARE_READ，那么这个文件在本次打开之后、关闭之前，别次打开试图以读权限打开，则被允许，可以成功打开。如果别次打开试图以写权限打开，则一定失败，返

回共享冲突。

同时，如果本次打开只用了 FILE_SHARE_READ，而之前这个文件已经被另一次打开用写权限打开着，那么本次打开一定失败，返回共享冲突。其中的逻辑关系貌似比较复杂，读者应耐心理解。

CreateDisposition 参数说明了这次打开的意图。可能的选择如下（请注意这些选择不能组合）：

- FILE_CREATE：新建文件。如果文件已经存在，则这个请求失败。
- FILE_OPEN：打开文件。如果文件不存在，则请求失败。
- FILE_OPEN_IF：打开或新建。如果文件存在，则打开；如果不存在，则失败。
- FILE_OVERWRITE：覆盖。如果文件存在，则打开并覆盖其内容；如果文件不存在，则这个请求返回失败。
- FILE_OVERWRITE_IF：新建或覆盖。如果要打开的文件已存在，则打开它，并覆盖其内容；如果不存在，则简单地新建文件。
- FILE_SUPERSEDE：新建或取代。如果要打开的文件已存在。则生成一个新文件替代之；如果不存在，则简单地生成新文件。

请联系上面的 IoStatusBlock 参数中 Information 的说明。

最后一个重要的参数是 CreateOptions。在惯常的编程中，笔者使用 FILE_NON_DIRECTORY_FILE| FILE_SYNCHRONOUS_IO_NONALERT。此时文件被同步地打开，而且打开的是文件（而不是目录。创建目录请用 FILE_DIRECTORY_FILE）。所谓同步地打开的意义在于，以后每次操作文件的时候，比如写入文件，调用 ZwWriteFile，在 ZwWriteFile 返回时，文件写操作已经得到了完成，而不会有返回 STATUS_PENDING（未决）的情况。在非同步文件的情况下，返回未决是常见的，此时文件请求没有完成，使用者需要等待事件来等待请求的完成。当然，好处是使用者可以先去做别的事情。

要同步打开，前面的 DesiredAccess 必须含有 SYNCHRONIZE。

此外还有一些其他的情况，比如不通过缓冲操作文件，希望每次读/写文件都是直接往磁盘上操作的。此时 CreateOptions 中应该带标记 FILE_NO_INTERMEDIATE_BUFFERING。带了这个标记后，请注意操作文件每次读/写都必须以磁盘扇区大小（最

常见的是 512 字节）对齐，否则会返回错误。

这个函数是如此的烦琐，以至于再多的文档也不如一个可以利用的例子。早期笔者调用这个函数往往因为参数设置不对而导致打开失败，非常渴望找到一个实际可以使用的参数的范例。下面举例如下：

```
// 要返回的文件句柄
HANDLE file_handle = NULL;
// 返回值
NTSTATUS status;
// 首先初始化含有文件路径的 OBJECT_ATTRIBUTES
OBJECT_ATTRIBUTES object_attributes;
UNICODE_STRING ufile_name = RTL_CONST_STRING(L"\\??\\C:\\a.dat");
InitializeObjectAttributes(
    &object_attributes,
    &ufile_name,
    OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE,
    NULL,
    NULL);
// 以 OPEN_IF 方式打开文件
status = ZwCreateFile(
    &file_handle,
    GENERIC_READ | GENERIC_WRITE,
    &object_attributes,
    &io_status,
    NULL,
    FILE_ATTRIBUTE_NORMAL,
    FILE_SHARE_READ,
    FILE_OPEN_IF,
    FILE_NON_DIRECTORY_FILE |
    FILE_RANDOM_ACCESS |
    FILE_SYNCHRONOUS_IO_NONALERT,
    NULL,
    0);
```

值得注意的是路径的写法，并不是像应用层一样直接写“C:\\a.dat”，而是写成了“\\??\\C:\\a.dat”。这是因为 ZwCreateFile 使用的是对象路径，“C:”是一个符号链接对象，符号链接对象一般都在“\\??\\”路径下。

这种文件句柄的关闭非常简单，调用 ZwClose 即可。内核句柄的关闭不需要和打开在同一进程中。示例如下：

```
ZwClose(file_handle);
```

5.1.3 文件读/写操作

打开文件之后，最重要的操作是对文件的读/写。读与写的方法是对称的，只是参数输入与输出的方向不同。读取文件内容一般用 ZwReadFile，写文件一般使用 ZwWriteFile。

```
NTSTATUS ZwReadFile(
    IN HANDLE  FileHandle,
    IN HANDLE  Event  OPTIONAL,
    IN PIO_APC_ROUTINE  ApcRoutine  OPTIONAL,
    IN PVOID  ApcContext  OPTIONAL,
    OUT PIO_STATUS_BLOCK  IoStatusBlock,
    OUT PVOID  Buffer,
    IN ULONG  Length,
    IN PLARGE_INTEGER  ByteOffset  OPTIONAL,
    IN PULONG  Key  OPTIONAL);
```

FileHandle：是前面 ZwCreateFile 成功后所得到的 FileHandle。如果是内核句柄，ZwReadFile 和 ZwCreateFile 并不需要在同一个进程中。句柄是各进程通用的。

Event：一个事件，用于异步完成读时。下面的举例始终用同步读，所以忽略这个参数，请始终填写 NULL。

ApcRoutine：回调例程，用于异步完成读时。下面的举例始终用同步读，所以忽略这个参数，请始终填写 NULL。

IoStatusBlock：返回结果状态。同 ZwCreateFile 中的同名参数。

Buffer：缓冲区。如果读文件的内容成功，则内容被读到这个缓冲区里。

Length：描述缓冲区的长度。这个长度也就是试图读取文件的长度。

ByteOffset：要读取的文件的偏移量，也就是要读取的内容在文件中的位置。一般地，不要设置为 NULL。文件句柄不一定支持直接读取当前偏移量。

Key：读取文件时使用的一种附加信息，一般不使用，设置为 NULL。

返回值：成功的返回值是 STATUS_SUCCESS。只要读取到任意多个字节（不管是否符合输入的 Length 的要求），返回值都是 STATUS_SUCCESS，即使试图读取的长度范围超出了文件本来的大小。但是，如果仅读取文件长度之外的部分，则返回 STATUS_END_OF_FILE。

ZwWriteFile 的参数与 ZwReadFile 完全相同。当然，除了读/写文件外，有的读者可能会问是否提供一个 ZwCopyFile 用来拷贝一个文件。这个要求未能被满足。如果有这个需求，这个函数必须自己来编写。下面是一个例子，用来拷贝一个文件，用到了 ZwCreateFile、ZwReadFile 和 ZwWrite 这 3 个函数。不过作为本节的例子，只举出 ZwReadFile 和 ZwWriteFile 的部分。

```
NTSTATUS MyCopyFile(
    PUNICODE_STRING target_path,
    PUNICODE_STRING source_path)
{
    // 源和目标的文件句柄
    HANDLE target = NULL,source = NULL;
    // 用来拷贝的缓冲区
    PVOID buffer = NULL;
    LARGE_INTEGER offset = { 0 };
    IO_STATUS_BLOCK io_status = { 0 };

    do {
        // 这里请用前一节说到的例子打开 target_path 和 source_path 所对应的
        // 句柄 target 和 source,并为 buffer 分配一个页面，也就是 4KB 的内存
        … …
        // 然后用一个循环来读取文件。每次从源文件中读取 4KB 内容，然后往
        // 目标文件中写入 4KB，直到拷贝结束为止
        while(1) {
            length = 4*1024;      // 每次读取 4KB
            // 读取旧文件。注意 status
            status = ZwReadFile (
                    source,NULL,NULL,NULL,
                    &my_io_status,buffer, length,&offset,
                    NULL);
            if(!NT_SUCCESS(status))
            {
                // 如果状态为 STATUS_END_OF_FILE，则说明文件
                // 的拷贝已经成功地结束了
                if(status == STATUS_END_OF_FILE)
                    status = STATUS_SUCCESS;
                break;
            }
            // 获得实际读取到的长度
            length = IoStatus.Information;

            // 现在读取了内容。读出的长度为 length，那么写入
            // 的长度也应该是 length。写入必须成功，如果失败，则返回错误
```

```
            status = ZwWriteFile(
                target,NULL,NULL,NULL,
                &my_io_status,
                buffer,length,&offset,
                NULL);
            if(!NT_SUCCESS(status))
                break;

            // offset 移动，然后继续，直到出现 STATUS_END_OF_FILE
            // 的时候才结束
            offset.QuadPart += length;
        }
    } while(0);

    // 在退出之前，释放资源，关闭所有的句柄
    if(target != NULL)
        ZwClose(target);
    if(source != NULL)
        ZwClose(source);
    if(buffer != NULL)
        ExFreePool(buffer);
    return STATUS_SUCCESS;
}
```

除了读/写之外，文件还有很多的操作，比如删除、重新命名、枚举。这些操作将在后面实例中用到时，再详细讲解。

5.2 注册表操作

5.2.1 注册表键的打开

与在应用程序中编程的方式类似，注册表是一个巨大的树形结构。操作一般都是打开某个子键，子键下有若干个值可以获得，每一个值有一个名字。值有不同的类型，一般需要查询才能获得其类型。

子键一般用一个路径来表示。与应用程序编程的一点重大不同是这个路径的写法不一样。一般应用编程中需要提供一个根子键的句柄，而驱动编程中则全部用路径表示，见表 5-1。

表 5-1

应用编程中对应的子键	驱动编程中的路径写法
HKEY_LOCAL_MACHINE	\Registry\Machine
HKEY_USERS	\Registry\User
HKEY_CLASSES_ROOT	没有对应的路径
HKEY_CURRENT_USER	没有简单的对应路径，但是可以求得

实际上，应用程序和驱动程序很大的一个不同在于应用程序总是由某个“当前用户”启动的，因此可以直接读取当前用户的 HKEY_CLASSES_ROOT 和 HKEY_CURRENT_USER。但是显然这种透明切换用户的功能没有做到内核里。

打开注册表键使用函数 ZwOpenKey，新建或者打开则使用 ZwCreateKey。一般在驱动编程中，使用 ZwOpenKey 的情况比较多见，下面以此为例进行讲解。ZwOpenKey 的原型如下：

```
NTSTATUS
    ZwOpenKey(
        OUT PHANDLE  KeyHandle,
        IN ACCESS_MASK  DesiredAccess,
        IN POBJECT_ATTRIBUTES  ObjectAttributes
    );
```

这个函数和 ZwCreateFile 是类似的，它并不接受直接传入一个字符串来表示一个子键，而是要求输入一个 OBJECT_ATTRIBUTES 的指针。如何初始化一个 OBJECT_ATTRIBUTES，请参考前面讲解 ZwCreateFile 的章节。

DesiredAccess 支持一系列的组合权限，可以是下面所有权限的任何组合。

- KEY_QUERY_VALUE：读取键下的值。
- KEY_SET_VALUE：设置键下的值。
- KEY_CREATE_SUB_KEY：生成子键。
- KEY_ENUMERATE_SUB_KEYS：枚举子键。

不过，实际上可以用 KEY_READ 来作为通用的读权限组合，这是一个组合宏。此外对应的有 KEY_WRITE。如果需要获得全部的权限，可以使用 KEY_ALL_ACCESS。

下面是一个例子，这个例子非常的有实用价值。它读取注册表中保存的 Windows 系统目录（指 Windows 目录）的位置。不过这里只涉及打开子键，并不读取值。读取具体的值在后面的小节中再完成。

Windows 目录的位置被称为 SystemRoot，这个值保存在注册表中，路径是“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion”。当然，请注意在驱动编程中的写法有所不同。下面的代码初始化一个 OBJECT_ATTRIBUTES。

```
HANDLE my_key = NULL;
NTSTATUS status;

    // 定义要获取的路径
UNICODE_STRING my_key_path =
    RTL_CONSTANT_STRING(
L"\\Registry\\Machine\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion");
OBJECT_ATTRIBUTE my_obj_attr = { 0 };

// 初始化 OBJECT_ATTRIBUTE
InitializeObjectAttributes(
    &my_obj_attr,
    &my_key_path,
    OBJ_CASE_INSENSITIVE,
    NULL,
    NULL);
// 接下来是打开 Key
status = ZwOpenKey(&my_key,KEY_READ,&my_obj_attr);
if(!NT_SUCCESS(status))
{
    // 失败处理
    ……
}
```

上面的代码得到了 my_key。子键已经打开，接下来的步骤是读取下面的 SystemRoot 值。这在后面的小节中讲述。

5.2.2 注册表值的读

一般使用 ZwQueryValueKey 来读取注册表中键的值。要注意的是，注册表中的值可能有多种数据类型，而且长度也是没有定数的。为此，在读取过程中，就可能要面对很多种可能的情况。ZwQueryValueKey 这个函数的原型如下：

```
NTSTATUS ZwQueryValueKey(
    IN HANDLE  KeyHandle,
    IN PUNICODE_STRING  ValueName,
    IN KEY_VALUE_INFORMATION_CLASS  KeyValueInformationClass,
    OUT PVOID  KeyValueInformation,
```

```
    IN ULONG  Length,
    OUT PULONG  ResultLength
    );
```

KeyHandle：这是用 ZwCreateKey 或者 ZwOpenKey 所打开的一个注册表键句柄。

ValueName：要读取的值的名字。

KeyValueInformationClass：本次查询所需要查询的信息类型。这有如下 3 种可能。

- KeyValueBasicInformation：获得基础信息，包含值名和类型。
- KeyValueFullInformation：获得完整信息，包含值名、类型和值的数据。
- KeyValuePartialInformation：获得局部信息，包含类型和值数据。

实际上，很容易看出名字是已知的，获得基础信息是多此一举。同样获得完整信息也是浪费内存空间，因为调用 ZwQueryValueKey 的目的是为了得到类型和值数据。因此使用 KeyValuePartialInformation 最常用。当采用 KeyValuePartialInformation 的时候，一个类型为 KEY_VALUE_PARTIAL_INFORMATION 的结构将被返回到参数 KeyValueInformation 所指向的内存中。

KeyValueInformation：当 KeyValueInformationClass 被设置为 KeyValuePartialInformation 时，KEY_VALUE_PARTIAL_INFORMATION 结构将被返回到这个指针所指的内存中。下面是结构 KEY_VALUE_PARTIAL_INFORMATION 的原型。

```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
    ULONG  TitleIndex;          // 请忽略这个成员
    ULONG  Type;                // 数据类型
    ULONG  DataLength;          // 数据长度
    UCHAR  Data[1];             // 可变长度的数据
}KEY_VALUE_PARTIAL_INFORMATION,*PKEY_VALUE_PARTIAL_INFORMATIO;
```

上面的数据类型 Type 有很多种可能，但是最常见的有以下几种：

- REG_BINARY：十六进制数据。
- REG_DWORD：四字节整数。
- REG_SZ：以空结束的 Unicode 字符串。

Length：用户传入的输出空间 KeyValueInformation 的长度。

ResultLength：返回实际需要的长度。

返回值：如果说实际需要的长度比 Length 要大，那么返回 STATUS_BUFFER_OVERFLOW 或者 STATUS_BUFFER_TOO_SMALL。如果成功读出了全部数据，那么返回 STATUS_SUCCESS。其他的情况，返回一个错误码。

下面请读者考虑如何把上一节的函数写完整。这其中比较常见的一个问题是在读取注册表键下的值之前，往往不知道这个值有多长。所以有些比较偷懒的程序员总是定义一个足够大小的空间（比如 512 字节）。这样的坏处是浪费内存（一般都是在堆栈中定义，而内核编程中堆栈空间被耗尽又是另一个常见的蓝屏问题）。此外也无法避免值实际上大于该长度的情况。为此应该耐心地首先获取长度，然后不足时再动态分配内存进行读取。下面是示例代码：

```
// 要读取的值的名字
UNICODE_STRING my_key_name =
    RTL_CONSTANT_STRING(L"SystemRoot");
// 用来试探大小的 key_infor
KEY_VALUE_PARTIAL_INFORMATION key_infor;
// 最后实际用到的 key_infor 指针。内存分配在堆中
PKEY_VALUE_PARTIAL_INFORMATION ac_key_infor;
ULONG ac_length;
……
// 前面已经打开了句柄 my_key，下面如此来读取值
status = ZwQueryValueKey(
    my_key,
    &my_key_name,
    KeyValuePartialInformation,
    &key_infor,
    sizeof(KEY_VALUE_PARTIAL_INFORMATION),
    &ac_length);
if(!NT_SUCCESS(status) &&
    status != STATUS_BUFFER_OVERFLOW &&
    status != STATUS_BUFFER_TOO_SMALL)
{
    // 错误处理
    …
}
// 如果没失败，那么分配足够的空间，再次读取
ac_key_infor = (PKEY_VALUE_PARTIAL_INFORMATION)
    ExAllocatePoolWithTag(NonpagedPool,ac_length ,MEM_TAG);
if(ac_key_infor == NULL)
{
    stauts = STATUS_INSUFFICIENT_RESOURCES;
    // 错误处理
```

```
    ...
}
status = ZwQueryValueKey(
    my_key,
    &my_key_name,
    KeyValuePartialInformation,
    ac_key_infor,
    ac_length,
    &ac_length);
// 到此为止，如果 status 为 STATUS_SUCCESS，则要读取的数据已经
// 在 ac_key_infor->Data 中。请利用前面学到的知识，转换为
// UNICODE_STRING
.....
```

5.2.3 注册表值的写

实际上注册表的写入比读取要简单，因为这省略了一个尝试定义数据的大小的过程，直接将数据写入即可。写入值一般使用函数 ZwSetValueKey。这个函数的原型如下：

```
NTSTATUS ZwSetValueKey(
    IN HANDLE  KeyHandle,
    IN PUNICODE_STRING  ValueName,
    IN ULONG  TitleIndex  OPTIONAL,
    IN ULONG  Type,
    IN PVOID  Data,
    IN ULONG  DataSize
    );
```

其中的 TileIndex 参数请始终填入 0。

KeyHandle、ValueName、Type 这 3 个参数和 ZwQueryValueKey 中对应的参数相同。不同的是 Data 和 DataSize，Data 是要写入的数据的开始地址，而 DataSize 是要写入的数据的长度。由于 Data 是一个空指针，因此，Data 可以指向任何数据。也就是说，不管 Type 是什么，都可以在 Data 中填写相应的数据写入。

使用 ZwSetValueKey 的时候，并不需要该 Value 已经存在。如果该 Value 已经存在，那么其值会被这次写入覆盖；如果不存在，则会新建一个。下面的例子写入一个名字为“Test”，而且值为“My Test Value”的字符串值。假设 my_key 是一个已经打开的子键的句柄。

```
UNICODE_STRING name = RTL_CONSTANT_STRING(L"Test");
PWCHAR value = { L"My Test Value" };
```

```
…
// 写入数据。数据长度之所以要将字符串长度加上 1，是为了把最后一个空结束符
// 写入。笔者不确定如果不写入空结束符会不会有错，有兴趣的读者请自己测试一下
status = ZwSetValueKey(my_key,
        &name,0,REG_SZ,value,(wcslen(value)+1)*sizeof(WCHAR));
if(!NT_SUCCESS(status))
{
    // 错误处理
    ……
}
```

关于注册表的操作就介绍到这里了。如果有进一步的需求，建议读者阅读 WDK 相关的文档。

第6章　时间与线程

本章的内容包括系统时间的获取、睡眠、内核线程和线程间同步等。获得系统时间和进行睡眠在应用编程和内核编程中都是常见的需求。此外，内核编程与应用程序编程一个很重要的不同点在于：在应用编程中，许多情况下只需要考虑单线程就可以了，除非有特殊的需求导致了多线程编程的应用。而在内核编程中，在绝大多数情况下，我们所写的代码都位于多线程环境中。因此，掌握线程间同步的编程，就变得非常重要。

6.1 时间与定时器

6.1.1 获得当前滴答数

获得系统日期和时间往往是为了写日志，获得启动毫秒数则很适合用来做一个随机数的种子。有时也使用时间相关的函数来寻找程序的性能瓶颈。

熟悉 Win32 应用程序开发的读者会知道有一个函数 GetTickCount()，这个函数返回系统自启动之后经历的毫秒数。在驱动开发中有一个对应的函数 KeQueryTickCount()，这个函数的原型如下：

```
VOID
    KeQueryTickCount(
        OUT PLARGE_INTEGER  TickCount
    );
```

遗憾的是，被返回到 TickCount 中的并不是一个简单的毫秒数。这是一个“滴答”数。但是一个“滴答”到底为多长的时间，在不同的硬件环境下可能有所不同。为此，必须结合另一个函数使用。下面这个函数获得一个“滴答”的具体的 100 纳秒数。

```
ULONG
    KeQueryTimeIncrement(
        );
```

得知以上的关系之后，下面的代码可以求得实际的毫秒数：

```
void MyGetTickCount (PULONG msec)
{
    LARGE_INTEGER tick_count;
    ULONG myinc = KeQueryTimeIncrement();
    KeQueryTickCount(&tick_count);
```

```
    tick_count.QuadPart *= myinc;
    tick_count.QuadPart /= 10000;
    *msec = tick_count.LowPart;
}
```

这不是一个简单的过程。不过幸运的是，现在有代码可以拷贝了。

6.1.2 获得当前系统时间

接下来的一个需求是得到当前的可以供人们理解的时间，包括年、月、日、时、分、秒这些要素。在驱动中不能使用诸如 CTime 之类的 MFC 类。不过与之对应的有 TIME_FIELDS，这个结构中含有对应的时间要素。

KeQuerySystemTime()得到当前时间，但是得到的并不是当地时间，而是一个格林威治时间。之后请使用 ExSystemTimeToLocalTime()转换成当地时间。这两个函数的原型如下：

```
VOID KeQuerySystemTime(
    OUT PLARGE_INTEGER  CurrentTime
    );
VOID ExSystemTimeToLocalTime(
    IN PLARGE_INTEGER  SystemTime,
    OUT PLARGE_INTEGER  LocalTime
    );
```

这两个函数使用的“时间”都是长长整型数据结构。这不是人们可以阅读的，必须通过函数 RtlTimeToTimeFields 转换为 TIME_FIELDS。这个函数原型如下：

```
VOID RtlTimeToTimeFields(
    IN PLARGE_INTEGER  Time,
    IN PTIME_FIELDS  TimeFields
    );
```

读者需要实际应用一下来加深印象。下面写出一个函数，这个函数返回一个字符串。这个字符串写出当前的年、月、日、时、分、秒，这些数字之间用“-”号隔开。这是一个很有用的函数，而且同时用到上面 3 个函数。此外，请读者回忆前面关于字符串打印的相关章节。

```
PWCHAR MyCurTimeStr()
{
    LARGE_INTEGER snow,now;
    TIME_FIELDS now_fields;
    static WCHAR time_str[32] = { 0 };
```

```
    // 获得标准时间
    KeQuerySystemTime(&snow);
    // 转换为当地时间
    ExSystemTimeToLocalTime(&snow,&now);
    // 转换为人们可以理解的时间要素
    RtlTimeToTimeFields(&now,&now_fields);
    // 打印到字符串中
    RtlStringCchPrintfW(
        time_str,
        32*2,
        L"%4d-%2d-%2d %2d-%2d-%2d",
        now_fields.Year,now_fields.Month,now_fields.Day,
        now_fields.Hour,now_fields.Minute,now_fields.Second);
    return time_str;
}
```

请注意 time_str 是静态变量，这使得这个函数不具备多线程安全性。请读者考虑一下，如何保证多个线程同时调用这个函数的时候，不出现冲突？

6.1.3 使用定时器

使用过 Windows 应用程序编程的读者一定对 SetTimer()印像尤深。当需要定时执行任务的时候，SetTimer()变得非常有用。这个功能在驱动开发中可以通过一些不同的替代方法来实现。比较经典的对应是 KeSetTimer()，这个函数的原型如下：

```
BOOLEAN
    KeSetTimer(
        IN PKTIMER  Timer,                  // 定时器
        IN LARGE_INTEGER  DueTime,          // 延后执行的时间
        IN PKDPC  Dpc  OPTIONAL             // 要执行的回调函数结构
    );
```

其中的定时器 Timer 和要执行的回调函数结构 Dpc 都必须先初始化，其中 Timer 的初始化比较简单。下面的代码可以初始化一个 Timer：

```
KTIMER my_timer;
KeInitializeTimer(&my_timer);
```

Dpc 的初始化比较麻烦，这是因为需要提供一个回调函数。初始化 Dpc 的函数原型如下：

```
VOID
KeInitializeDpc(
    IN PRKDPC  Dpc,
```

```
    IN PKDEFERRED_ROUTINE  DeferredRoutine,
    IN PVOID  DeferredContext
);
```

PKDEFERRED_ROUTINE 这个函数指针类型所对应的函数类型实际上是这样的：

```
VOID
CustomDpc(
    IN struct _KDPC  *Dpc,
    IN PVOID  DeferredContext,
    IN PVOID  SystemArgument1,
    IN PVOID  SystemArgument2
);
```

读者需要关心的只是 DeferredContext。这个参数是 KeInitializeDpc 调用时传入的参数，用来提供给 CustomDpc 被调用的时候，让用户传入一些参数。

至于后面的 SystemArgument1 和 SystemArgument2 则不要理会。Dpc 是回调这个函数的 KDPC 结构。

请注意这是一个“延时执行”的过程，而不是一个定时执行的过程。因此每次执行了之后，下次就不会再被调用了。如果想要定时反复执行，就必须在每次 CustomDpc 函数被调用的时候，再次调用 KeSetTimer，来保证下次还可以执行。

值得注意的是，CustomDpc 将运行在 APC 中断级，因此并不是所有的事情都可以做（在调用任何内核系统函数的时候，请注意 WDK 说明文档中标明的中断级要求）。

关于中断级

内核的代码始终运行在某个“中断级”上。读者应该了解代码并非从头到尾地连续执行，在某些执行过程中随时可能被打断：异常发生、中断发生、线程切换等。这使内核的开发者随时必须清楚，哪些代码可能被哪些代码打断。

为此产生了“中断级”的概念。高中断级上运行的代码不会被低中断级上运行的代码中断。举个简单的例子：一个请求的完成函数往往在 Dispatch 级（请参阅“7.2 请求处理”一节）。而一个系统线程中的代码一般运行在 Passive 级。Dispatch 级比 Passive 级别更高，所以你不用担心前者中的代码会忽然中断切换到后者。主要的中断级从高到低是 Dispatch > APC > Passive。

在帮助文档中每个内核 API 都有对应的说明，这个函数应该在什么中断级范围内使用，所以使用前总是请注意文档上的说明。

因为那些操作非常烦琐，因此要完全实现定时器的功能，最好自己封装一些东西。下面的结构封装了全部需要的信息：

```
// 内部时钟结构
typedef struct MY_TIMER_
{
    KDPC dpc;
    KTIMER timer;
    PKDEFERRED_ROUTINE func;
    PVOID private_context;
} MY_TIMER,*PMY_TIMER;

// 初始化这个结构
void MyTimerInit(PMY_TIMER timer, PKDEFERRED_ROUTINE func)
{
    // 请注意，笔者把回调函数的上下文参数设置为 timer,为什么要这样做呢
    KeInitializeDpc(&timer->dpc,sf_my_dpc_routine,timer);
    timer->func = func;
    KeInitializeTimer(&timer->timer);
    return (wd_timer_h)timer;
}

// 让这个结构中的回调函数在 n 毫秒之后开始运行
BOOLEAN MyTimerSet(PMY_TIMER timer,ULONG msec,PVOID context)
{
    LARGE_INTEGER due;
    // 注意时间单位的转换。这里 msec 是毫秒
    due.QuadPart = -10000*msec;
    // 用户私有上下文
    timer->private_context = context;
    return KeSetTimer(&timer->timer,due,&mytimer->dpc);
};

// 停止执行
VOID MyTimerDestroy(PMY_TIMER timer)
{
    KeCancelTimer(&mytimer->timer);
};
```

使用结构 PMY_TIMER 已经比结合使用 KDPC 和 KTIMER 简便许多，但是还是有一些要注意的地方。在真正的 OnTimer 回调函数中要获得上下文，必须要从 timer->private_context 中获得。此外，OnTimer 中还有必要再次调用 MyTimerSet()，来保证下次依然得到执行。

```
VOID
MyOnTimer (
        IN struct _KDPC  *Dpc,
        IN PVOID  DeferredContext,
        IN PVOID  SystemArgument1,
        IN PVOID  SystemArgument2
    )
{
    // 这里传入的上下文是 timer 结构，用来下次再启动延时调用
    PMY_TIMER timer = (PMY_TIMER)DeferredContext;
    // 获得用户上下文
    PVOID my_context = timer->private_context;

    // 在这里做 OnTimer 中要做的事情
    ……
    // 再次调用。这里假设每 1 秒执行一次
    MyTimerSet(timer,1000,my_context);
};
```

关于定时器就介绍到这里了。

6.2 线程与事件

6.2.1 使用系统线程

有时候需要使用线程来完成一个或者一组任务，这些任务可能耗时过长，而开发者又不想让当前系统停止下来等待。在驱动中停止等待很容易使整个系统陷入“停顿”，最后可能只能重启电脑。但一个单独的线程长期等待，还不至于对系统造成致命的影响。另一些任务是希望长期、不断地执行，比如不断地写入日志，为此启动一个特殊的线程来执行它们是最好的方法。

在驱动中生成的线程一般是系统线程。系统线程所在的进程名为“System”，用到的内核 API 函数原型如下：

```
NTSTATUS
    PsCreateSystemThread(
    OUT PHANDLE  ThreadHandle,
    IN ULONG  DesiredAccess,
    IN POBJECT_ATTRIBUTES  ObjectAttributes  OPTIONAL,
```

```
    IN HANDLE  ProcessHandle  OPTIONAL,
    OUT PCLIENT_ID  ClientId  OPTIONAL,
    IN PKSTART_ROUTINE  StartRoutine,
    IN PVOID  StartContext);
```

这个函数的参数也很多，作者的使用经验如下：ThreadHandle 用来返回句柄，放入一个句柄指针即可。DesiredAccess 总是填写 0；后面 3 个参数都填写 NULL；最后的2个参数一个用于该线程启动的时候执行的函数，另一个用于传入该函数的参数。

下面要关心的就是那个启动函数的原型。这个原型比起定时器回调函数倒是异常的简单，没有任何多余的东西：

```
VOID CustomThreadProc(IN PVOID context)
```

可以传入一个参数，就是那个 context，context 就是 PsCreateSystemThread 中的 StartContext。值得注意的是，线程的结束应该在线程中自己调用 PsTerminateSystemThread 来完成，此外得到的句柄也必须要用 ZwClose 来关闭。但是请注意：关闭句柄并不结束线程。

下面举一个例子。这个例子传递一个字符串指针到一个线程中打印一下，然后结束该线程。当然打印字符串这种事情没有必要单独开一个线程来做，这里只是一个简单的示例。请注意，这个代码中有一个隐藏的错误，请读者指出这个错误是什么。

```
// 我的线程函数。传入一个参数，这个参数是一个字符串
VOID MyThreadProc(PVOID context)
{
    PUNICODE_STRING str = (PUNICODE_STRING)context;
    // 打印字符串
    KdPrint(("PrintInMyThread:%wZ\r\n",str));
    // 结束自己
    PsTerminateSystemThread(STATUS_SUCCESS);
}

VOID MyFunction()
{
    UNICODE_STRING str = RTL_CONSTANT_STRING(L"Hello!");
    HANDLE thread = NULL;
    NTSTATUS status;

    status = PsCreateSystemThread(
        &thread,0L,NULL,NULL,NULL,MyThreadProc,(PVOID)&str);
    if(!NT_SUCCESS(status))
    {
        // 错误处理
        ...
```

```
    }
    // 如果成功了，可以继续做自己的事。之后得到的句柄要关闭
    ZwClose(thread);
}
```

以上错误之处在于：MyThreadProc 执行的时候，MyFunction 可能已经执行完毕了。执行完毕之后，堆栈中的 str 已经无效，此时再执行 KdPrint 去打印 str 一定会蓝屏。这也是一个非常隐蔽，但是非常容易犯的错误。

合理的方法是在堆中分配 str 的空间，或者 str 必须在全局空间中。请读者自己写出正确的方法。

但是读者会发现，以上的写法在正确的代码中也是常见的。原因是这样做的时候，在 PsCreateSystemThread 结束之后，开发者会在后面加上一个等待线程结束的语句。

这样就没有任何问题了，因为在这个线程结束之前，这个函数都不会执行完毕，所以栈内存空间不会失效。

这样做的目的一般不是为了让任务并发，而是为了利用线程上下文环境而做的特殊处理，比如防止重入等。在后面的章节读者会学到这方面的技巧。

如何等待线程结束，在后面 6.2.3 节“使用同步事件”中进一步讲述。

6.2.2 在线程中睡眠

许多读者一定使用过 Sleep 函数，这能使程序停下一段时间。许多需要连续、长期执行，但是又不希望占太多 CPU 使用率的任务，可以在中间加入睡眠。这样能使 CPU 使用率大大降低，即使睡眠的时间非常短（几十毫秒）。

在驱动中也可以睡眠。使用到的内核函数的原型如下：

```
NTSTATUS
KeDelayExecutionThread(
IN KPROCESSOR_MODE  WaitMode,
IN BOOLEAN  Alertable,
IN PLARGE_INTEGER  Interval);
```

这个函数的参数简单明了。WaitMode 请总是填写 KernelMode，因为现在是在内核编程中使用。Alertable 表示是否允许线程报警（用于重新唤醒），但是目前没有必要用到这么高级的功能，请总是填写 FALSE。剩下的就是 Interval 了，表明要睡眠多久。

但是这个看似简单的参数说明起来却异常的复杂，为此建议读者使用下面简单的睡眠函数，这个函数可以指定睡眠多少毫秒，而没有必要自己去换算时间（这个函数中有

睡眠时间的转换)。

```
#define DELAY_ONE_MICROSECOND    (-10)
#define DELAY_ONE_MILLISECOND    (DELAY_ONE_MICROSECOND*1000)
VOID MySleep(LONG msec)
{
    LARGE_INTEGER my_interval;
    my_interval.QuadPart = DELAY_ONE_MILLISECOND;
    my_interval.QuadPart *= msec;
    KeDelayExecutionThread(KernelMode,0,&my_interval);
}
```

当然要睡眠几秒也是可以的，1 毫秒为千分之一秒，所以乘以 1000 就可以表示秒数。

在一个线程中用循环进行睡眠，也可以实现一个自己的定时器。考虑前面说的定时器的缺点：中断级较高，有一些事情不能做。在线程中用循环睡眠，每次睡眠结束之后调用自己的回调函数，也可以起到类似的效果。而且系统线程执行中是 Passive 中断级，睡眠之后依然是这个中断级，所以不像前面提到的定时器那样有限制。

请读者自己写出用线程+睡眠来实现定时器的例子。

6.2.3 使用同步事件

一些读者可能熟悉“事件驱动”编程技术，但是这里的“事件”与之不同。内核中的事件是一个数据结构，这个结构的指针可以当做一个参数传入一个等待函数中。如果这个事件不被“设置”，则这个等待函数不会返回，这个线程被阻塞；如果这个事件被“设置”，则等待结束，可以继续下去。

这常常用于多个线程之间的同步。如果一个线程需要等待另一个线程完成某事后才能做某事，则可以使用事件等待。另一个线程完成后设置事件即可。

这个数据结构是 KEVENT，读者没有必要去了解其内部结构，这个结构总是用 KeInitlizeEvent 初始化。这个函数原型如下：

```
VOID
KeInitializeEvent(
    IN PRKEVENT  Event,
    IN EVENT_TYPE  Type,
    IN BOOLEAN  State
);
```

第一个参数是要初始化的事件；第二个参数是事件类型，详见于后面的解释；第三个参数是初始化状态，一般地设置为 FALSE，也就是未设状态，这样等待者需要等待

设置之后才能通过。

事件不需要销毁。

设置事件使用函数 KeSetEvent。这个函数原型如下：

```
LONG
KeSetEvent(
    IN PRKEVENT  Event,
    IN KPRIORITY  Increment,
    IN BOOLEAN  Wait
    );
```

Event 是要设置的事件；Increment 用于提升优先权，目前设置为 0 即可；Wait 表示是否后面马上紧接着一个 KeWaitSingleObject 来等待这个事件，一般设置为 TRUE。（事件初始化之后，一般就要开始等待了。）

使用事件的简单代码如下：

```
// 定义一个事件
KEVENT event;
// 事件初始化
KeInitializeEvent(&event,SynchronizationEvent,TRUE);
……
// 事件初始化之后就可以使用了。在一个函数中，你可以等待某
// 个事件。如果这个事件没有被人设置，那就会阻塞在这里继续等待
KeWaitForSingleObject(&event,Executive,KernelMode,0,0);
……

// 这是另一个地方，有人设置这个事件。只要一设置这个事件
// 前面等待的地方，将继续执行
KeSetEvent(&event);
```

由于在 KeInitializeEvent 中使用了 SynchronizationEvent，导致这个事件成为所谓的“自动重设”事件。一个事件如果被设置，那么所有 KeWaitForSingleObject 等待这个事件的地方都会通过。如果要能继续重复使用这个事件，必须重设（Reset）这个事件。当 KeInitializeEvent 中第二个参数被设置为 NotificationEvent 的时候，这个事件必须要手动重设才能使用。手动重设使用函数 KeResetEvent。

```
LONG
KeResetEvent(
    IN PRKEVENT  Event
);
```

如果这个事件初始化的时候是 SynchronizationEvent 事件，那么只有一个线程的 KeWaitForSingleObject 可以通过，通过之后被自动重设，其他的线程就只能继续等待了。

这可以起到一个同步作用，所以叫做同步事件。不能起到同步作用的是通知事件（NotificationEvent）。请注意不能用手工设置通知事件的方法来取代同步事件。请读者思考一下这是为什么。

回忆 6.2.1 节“使用系统线程”最后的例子。在那里曾经有一个需求：就是等待线程中的函数 KdPrint 结束之后，外面生成线程的函数再返回。这可以通过一个事件来实现：线程中打印结束之后，设置事件，外面的函数再返回。为了编码简单，笔者使用了一个静态变量做事件，这种方法在线程同步中用得极多，请务必熟练掌握。

```
static KEVENT s_event;

// 我的线程函数。传入一个参数，这个参数是一个字符串
VOID MyThreadProc(PVOID context)
{
    PUNICODE_STRING str = (PUNICODE_STRING)context;
    KdPrint(("PrintInMyThread:%wZ\r\n",str));
    KeSetEvent(&s_event);          // 在这里设置事件
    PsTerminateSystemThread(STATUS_SUCCESS);
}

// 生成线程的函数
VOID MyFunction()
{
    UNICODE_STRING str = RTL_CONSTANT_STRING(L"Hello!");
    HANDLE thread = NULL;
    NTSTATUS status;

    KeInitializeEvent(&event,SynchronizationEvent,TRUE);  // 初始化事件
    status = PsCreateSystemThread(
        &thread,0L,NULL,NULL,NULL,MyThreadProc,(PVOID)&str);
    if(!NT_SUCCESS(status))
    {
        // 错误处理
        ...
    }
    ZwClose(thread);
    // 等待事件结束再返回
    KeWaitForSingleObject(&s_event,Executive,KernelMode,0,0);
}
```

实际上，等待线程结束并不一定要用事件，线程本身也可以当做一个事件来等待，但是这里为了演示事件的用法而使用了事件。以上的方法调用线程则不必担心 str 的内存空间会无效了，因为这个函数在线程执行完 KdPrint 之后才返回。缺点是这个函数不能起到并发执行的作用。

第7章　驱动、设备与请求

7.1 驱动与设备

7.1.1 驱动入口与驱动对象

驱动开发程序员所编写的驱动程序对应一个结构，这个结构名为 DRIVER_OBJECT，对应一个“驱动程序”。下面的代码展示的是一个最简单的驱动程序。

```
#include <ntddk.h>
NTSTATUS
    DriverEntry (
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING  RegistryPath
)
{
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    return status;
}
```

函数 DriverEntry 是每个驱动程序中必需的，如同 Win32 应用程序里的 WinMain。DriverEntry 的第一个参数就是一个 DRIVER_OBJECT 的指针，这个 DRIVER_OBJECT 结构就对应当前编写的驱动程序，其内存是 Windows 系统已经分配的。

第二个参数 RegistryPath 是一个字符串，代表一个注册表子键。这个子键是专门分配给这个驱动程序使用的，用于将驱动配置信息保存到注册表中。至于读写注册表的方法，请参照前面章节中的内容。

当读者打算反汇编阅读一个内核程序的时候，可先寻找 DriverEntry 的位置。

DriverEntry 的返回值决定这个驱动的加载是否成功。如果返回为 STATUS_SUCCESS，则驱动将成功加载；否则，驱动加载失败。

7.1.2 分发函数和卸载函数

DRIVER_OBJECT 中含有分发函数指针，这些函数用来处理发送到这个驱动的各种请求。Windows 总是自己调用 DRIVER_OBJECT 下的分发函数来处理这些请求，所以

编写一个驱动程序，实质上就是自己编写这些处理请求的分发函数。

DRIVER_OBJECT 下的分发函数指针的个数为 IRP_MJ_MAXIMUM_FUNCTION，保存在一个数组中。下面的代码设置所有分发函数的地址为同一个函数。

```
NTSTATUS
DriverEntry (
    IN PDRIVER_OBJECT   DriverObject,
    IN PUNICODE_STRING  RegistryPath
)
{
    ULONG i;
    for(i=0;i<IRP_MJ_MAXIMUM_FUNCTION;++i)
    {
        DriverObject->MajorFunctions[i] = MyDispatchFunction;
    }
    ...
}
```

这个设置固然不难，难的工作都在编写 MyDispatchFunction 这个函数上。因为所有的分发函数指针都指向这一个函数，那么这个函数当然要完成本驱动所有的功能了。下面是这个函数的原型。这个原型是 Windows 驱动编程的规范，不能更改。

```
NTSTATUS MyDispatchFunction(PDEVICE_OBJECT device,PIRP irp)
{
    ......
}
```

这里出现了 DEVICE_OBJECT 和 IRP 这两大结构。前一个表示一个由本驱动生成的设备对象；后一个表示一个系统请求。也就是说，现在要处理的是，发送给设备 device 的请求 irp。这两个结构在后面再进一步描述。

还有一个不放在分发函数数组中的函数，称为卸载函数，也非常重要。如果存在这个函数，则该驱动程序可以动态卸载。在卸载时，该函数会被执行。该函数原型如下：

```
VOID MyDriverUnload(PDRIVER_OBJECT driver)
{
    ......
}
```

这个函数的地址设置到 DriverObject->DriverUnload 即可。

由于没有返回值，所以实际上在 DriverUnload 中，已经无法决定这个驱动能否卸载，只能做善后处理。

7.1.3 设备与符号链接

驱动程序和系统其他组件之间的交互是通过给设备发送或者接收发送给设备的请求来交互的。换句话说，一个没有任何设备的驱动是不能按规范方式和系统交互的，当然也不会收到任何 IRP，分发函数也失去了意义。

但并不意味着这样的驱动程序不存在。如果一个驱动程序只是想写写日志文件、Hook 某些内核函数或者做一些其他的小动作，也可以不生成任何设备，也不需要关心分发函数的设置。

如果驱动程序要和应用程序之间通信，则应该生成设备。此外还必须为设备生成应用程序可以访问的符号链接。下面的驱动程序生成了一个设备，并设置了分发函数。

```
#include <ntifs.h>   // 之所以用 ntifs.h 而不是 ntddk.h，是因为笔者习惯开发文件
                     // 系统驱动，实际上目前对读者来说这两个头文件没区别
NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg_path)
{
    NTSTATUS status;
    PDEVICE_OBJECT device;
    // 设备名
    UNICODE_STRING device_name =
        RTL_CONSTANT_STRING("\\Device\\MyCDO");
    // 符号链接名
    UNICODE_STRING symb_link =
        RTL_CONSTANT_STRING("\\DosDevices\\MyCDOSL");

    // 生成设备对象
    status = IoCreateDevice(
            driver,
            0,
            device_name,
            FILE_DEVICE_UNKNOWN,
            0,
            FALSE,
            &device);

    // 如果不成功，就返回
    if(!NT_SUCCESS(status))
        return status;
```

```
    // 生成符号链接
    status = IoCreateSymbolicLink(
        &symb_link,
        &device_name);
    if(!NT_SUCCESS(status))
    {
        IoDeleteDevice(device);
        return status;
    }
    // 设备生成之后，打开初始化完成标记
    device->Flags &= ~DO_DEVICE_INITIALIZING;
    return status;
}
```

这个驱动成功加载之后，生成一个名叫“\Device\MyCDO”的设备，然后再给这个设备生成一个符号链接，名字叫做“\DosDevices\MyCDOSL”。应用层可以通过打开这个符号链接来打开设备。应用层可以调用 CreateFile 就像打开文件一样打开，只是路径应该是“"\\.\ MyCDOSL”。前面的“\\.\”意味着后面是一个符号链接名，而不是一个普通的文件。请注意，由于 C 语言中斜杠要双写，所以正确的写法应该是“\\\\.\\”。与应用层交互的例子在“7.2 请求处理”一节中介绍。

7.1.4 设备的安全创建

上一节的例子只是简单的例子，很多情况下那些代码会不起作用。为了避免读者在实际编程中受到特殊情况的困扰，下面详细说明生成设备和符号链接需要注意的地方。生成设备的函数原型如下：

```
NTSTATUS
IoCreateDevice(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,
    IN BOOLEAN  Exclusive,
    OUT PDEVICE_OBJECT  *DeviceObject
    );
```

这个函数的参数也非常复杂，但是实际上需要注意的并不多。

第一个参数 DriverObject 是生成这个设备的驱动对象。

第二个参数 DeviceExtensionSize 非常重要。由于分发函数中得到的总是设备的指针，当用户需要在每个设备上记录一些额外的信息（比如用于判断这个设备是哪个设备的信息，以及不同的实际设备所需要记录的实际信息，如网卡上数据包的流量、过滤器所绑定真实设备指针等）时，需要指定设备扩展区内存的大小。如果 DeviceExtensionSize 设置为非 0，IoCreateDevice 会分配这个大小的内存在 DeviceObject->DeviceExtension 中，以后用户就可以从 DeviceObject->DeviceExtension 中来获得这些预先保存的信息了。

DeviceName 如前例，是设备的名字。目前生成设备，请总是生成在\Device\目录下，所以前面写的名字是“\Device\MyCDO”。其他路径也是可以的，但是这在本书描述范围之外。

DeviceType 表示设备类型。目前的范例无所谓的设备类型，所以填写 FILE_DEVICE_UNKNOWN 即可。

DeviceCharacteristics 目前请简单地填写 0 即可。

Exclusive 这个参数必须设置为 FALSE。文档没有做任何解释。

最后生成的设备对象指针返回到 DeviceObject 中。

这种设备生成之后，必须有系统权限的用户才能打开（比如管理员）。所以如果编程者写了一个普通的用户态的应用程序去打开这个设备进行交互，那么在很多情况下是可以的（用管理员登录的时候），但是偶尔又不行（用普通用户登录的时候）。结果困扰很久，其实是权限问题。

为了保证交互的成功与安全性，应该用服务程序与之交互。

但是依然有时候必须用普通用户打开设备。为了这个目的，设备必须是对所有的用户开放的。此时不能用 IoCreateDevice，必须用 IoCreateDeviceSecure。这个函数的原型如下：

```
NTSTATUS
IoCreateDeviceSecure(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,
    IN BOOLEAN  Exclusive,
    IN PCUNICODE_STRING  DefaultSDDLString,
    IN LPCGUID  DeviceClassGuid,
```

```
    OUT PDEVICE_OBJECT  *DeviceObject
)
```

这个函数增加了两个参数（其他的没变）。一个是 DefaultSDDLString，这是一个用于描述权限的字符串。描述这个字符串的格式需要大量的篇幅，但是没有这个必要，字符串“D:P(A;;GA;;;WD)”将满足“人人皆可以打开”的需求。

另一个参数是一个设备的 GUID。请随机手写一个 GUID，不要和其他设备的 GUID 冲突（不要复制、粘贴即可）。

下面是例子：

```
// 随机手写一个 GUID
const GUID DECLSPEC_SELECTANY MYGUID_CLASS_MYCDO =
{0x26e0d1e0L, 0x8189, 0x12e0, {0x99,0x14, 0x08, 0x00, 0x22, 0x30, 0x19,
0x03}};
// 全用户可读/写权限
UNICODE_STRING sddl =
    RLT_CONSTANT_STRING(L"D:P(A;;GA;;;WD)");
// 生成设备
status = IoCreateDeviceSecure( DriverObject,
                0,
                &device_name,
                FILE_DEVICE_UNKNOWN,
                FILE_DEVICE_SECURE_OPEN,
                FALSE,
                &sddl,
                (LPCGUID)&SFGUID_CLASS_MYCDO,
                &device);
```

使用这个函数的时候，必须连接库 wdmsec.lib。

7.1.5　设备与符号链接的用户相关性

从前面的例子来看，符号链接的命名貌似很简单。简单的符号链接（之所以称为简单，是因为还有一种使用 GUID 的符号链接，这在本书讨论范围之外）总是命名在\DosDevices\之下，但是实际上这会存在一些问题。

比较高级的 Windows 系统（哪个版本的操作系统很难讲，可能必须判定补丁号），符号链接也带有用户相关性。换句话说，如果一个普通用户创建了符号链接“\DosDevices\MyCDOSL”，那么，其实其他的用户是看不见这个符号链接的。

但是读者又会发现，如果在 DriverEntry 中生成符号链接，则所有用户都可以看见。原因是 DriverEntry 总是在进程“System”中执行。系统用户生成的符号链接是大家都可以看见的。

当前用户总是取决于当前启动当前进程的用户。实际编程中并不一定要在 DriverEntry 中生成符号链接，一旦在一个不明用户环境下生成符号链接，就可能出现注销然后换用户登录之后，符号链接“不见了”的严重错误。这也是常常让初学者抓狂几周都不知道如何解决的一个问题。

其实解决的方案很简单，任何用户都可以生成全局符号链接，让所有其他用户都能看见。将路径“\DosDevices\MyCDOSL”改为“\DosDevices\Global\MyCDOSL”即可。

但是在不支持符号链接用户相关性的系统上，生成“\DosDevices\Global\MyCDOSL”这样的符号链接是一种错误，为此必须先判断一下。这个判断虽然并不难，但是却全凭 WDK 帮助中一段代码的点拨（否则谁会想到 IoIsWdmVersionAvailable(1, 0x10)是什么意思呢？）。下面是这个例子，这个例子生成的符号链接总是随时可以使用，不用担心用户注销。

```
    UNICODE_STRING device_name;
    UNICODE_STRING symbl_name;
    if (IoIsWdmVersionAvailable(1, 0x10))
    {
        // 如果是支持符号链接用户相关性的版本的系统，用\DosDevices\Global
        RtlInitUnicodeString(&symbl_name,
L"\\DosDevices\\Global\\SymbolicLinkName");
    }
    else
    {
        // 如果是不支持的，则用\DosDevices
        RtlInitUnicodeString(&symbl, L"\\DosDevices\\SymbolicLinkName");
    }
    // 生成符号链接
    IoCreateSymbolicLink(&symbl_name, &device_name);
```

7.2 请求处理

7.2.1 IRP 与 IO_STACK_LOCATION

开发一个驱动有可能要处理各种 IRP。但是在本书范围内，只处理为了应用程序和驱动交互而产生的 IRP。IRP 的结构非常复杂，但是目前的需求下没有必要去深究它。

应用程序为了和驱动通信，首先必须打开设备，然后发送或者接收信息，最后关闭它。这至少需要 3 个 IRP：第一个是打开请求；第二个是发送或者接收信息；第三个是关闭请求。

IRP 的种类取决于主功能号。主功能号就是前面所说的 DRIVER_OBJECT 中分发函数指针数组中的索引。打开请求的主功能号是 IRP_MJ_CREATE，而关闭请求的主功能号是 IRP_MJ_CLOSE。

如果写有独立的处理 IRP_MJ_CREATE 和 IRP_MJ_CLOSE 的分发函数，就没有必要自然判断 IRP 的主功能号了。如果像前面的例子一样，使用一个函数处理所有的 IRP，那么首先就要得到 IRP 的主功能号。IRP 的主功能号在 IRP 的当前栈空间中。

IRP 总是发送给一个设备栈，到每个设备上的时候拥有一个“当前栈空间”来保存在这个设备上的请求信息。读者请暂时忽略这些细节。下面的代码在 MyDispatch 中获得主功能号，同时展示了几个常见的主功能号。

```
NTSTATUS MyDispatchFunction(PDEVICE_OBJECT device,PIRP irp)
{
    // 获得当前 IRP 调用栈空间
    PIO_STACK_LOCATION irpsp = IoGetCurrentIrpStackLocation(irp);
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    swtich(irpsp->MajorFunction)
    {
    // 处理打开请求
    case IRP_MJ_CREATE:
        ……
        break;
    // 处理关闭请求
    case IRP_MJ_CLOSE:
        ……
        break;
    // 处理设备控制信息
    case IRP_MJ_DEVICE_CONTROL:
        ……
        break;
    // 处理读请求
    case IRP_MJ_READ:
        ……
        break;
    // 处理写请求
    case IRP_MJ_WRITE:
        ……
```

```
            break;
        default:
            …
            break;
        }
        return status;
    }
```

用于与应用程序通信时，上面这些请求都由应用层 API 引发。对应的关系大致如表 7-1 所示。

表 7-1

应用层调用的 API	驱动层收到的 IRP 主功能号
CreateFile	IRP_MJ_CREATE
CloseHandle	IRP_MJ_CLOSE
DeviceIoControl	IRP_MJ_DEVICE_CONTROL
ReadFile	IRP_MJ_READ
WriteFile	IRP_MJ_WRITE

在了解以上信息的情况下，完成相关 IRP 的处理，就可以实现应用层和驱动层的通信了。具体的编程在后面紧接的两个小节里完成。

7.2.2 打开与关闭请求的处理

如果不能成功打开，则通信无法实现。要成功打开，只需要简单地返回成功就可以了。在一些有同步限制的驱动中（比如每次只允许一个进程打开设备）编程要更加复杂一点，但是现在忽略这些问题，暂时认为我们生成的设备任何进程都可以随时打开，不需要担心和其他进程冲突的问题。

简单地返回一个 IRP 成功（或者直接失败）是三部曲，如下：

（1）设置 irp->IoStatus.Information 为 0。关于 Information 的描述，请联系前面关于 IO_STATUS_BLOCK 结构的解释。

（2）设置 irp->IoStatus.Status 的状态。如果成功则设置 STATUS_SUCCESS，否则设置错误码。

（3）调用 IoCompleteRequest (irp,IO_NO_INCREMENT)，这个函数完成 IRP。

以上三步完成后，直接返回 irp->IoStatus.Status 即可。示例代码如下，这个函数能

完成打开和关闭请求。

```
NTSTATUS
MyCreateClose(
    IN PDEVICE_OBJECT device,
    IN PIRP           irp)
{
        irp->IoStatus.Information = 0;
    irp->IoStatus.Status = STATUS_SUCCESS;
        IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

当然，在前面设置分发函数的时候，应该加上：

```
DriverObject->MajorFunctions[IRP_MJ_CREATE] = MyCreateClose;
DriverObject->MajorFunctions[IRP_MJ_CLOSE] = MyCreateClose;
```

在应用层，打开和关闭这个设备的代码如下：

```
HANDLE device=CreateFile("\\\\.\\MyCDOSL",
    GENERIC_READ|GENERIC_WRITE,0,0,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_SYSTEM,0);
if (device == INVALID_HANDLE_VALUE)
{
    // …. 打开失败，说明驱动没加载，报错即可
}

// 关闭
CloseHandle(device);
```

7.2.3　应用层信息传入

应用层传入信息的时候，可以使用 WriteFile，也可以使用 DeviceIoControl。DeviceIoControl 是双向的，在读取设备的信息时也可以使用。因此本书以 DeviceIoControl 为例子进行说明。DeviceIoControl 称为设备控制接口，其特点是可以发送一个带有特定控制码的 IRP，同时提供输入和输出缓冲区。应用程序可以定义一个控制码，然后把相应的参数填写在输入缓冲区中，同时可以从输出缓冲区得到返回的更多信息。

当驱动得到一个 DeviceIoControl 产生的 IRP 的时候，需要了解的有当前的控制码、输入缓冲区的位置和长度，以及输出缓冲区的位置和长度。其中控制码必须预先用一个宏定义。定义的示例如下：

```
#define MY_DVC_IN_CODE \
    (ULONG)CTL_CODE(FILE_DEVICE_UNKNOWN, \
        0xa01, \
        METHOD_BUFFERED, \
        FILE_READ_DATA|FILE_WRITE_DATA)
```

其中 0xa01 这个数字是用户可以自定义的，其他的参数请照抄。

下面是获得这 3 个要素的例子。

```
NTSTATUS MyDeviceIoControl(
    PDEVICE_OBJECT  dev,
    PIRP  irp)
{
    // 得到 irpsp 的目的是为了得到功能号、输入/输出缓冲长度等信息
    PIO_STACK_LOCATION irpsp =
        IoGetCurrentIrpStackLocation(irp);
    // 首先要得到功能号
    ULONG code = irpsp->Parameters.DeviceIoControl.IoControlCode;
    // 得到输入/输出缓冲长度
    ULONG in_len =
        irpsp->Parameters.DeviceIoControl.InputBufferLength;
    ULONG out_len =
        irpsp->Parameters.DeviceIoControl.OutputBufferLength;
    // 请注意输入/输出缓冲是公用内存空间的
    PVOID buffer = irp->AssociatedIrp.SystemBuffer;

    // 如果是符合定义的控制码，处理完后返回成功
    if(code == MY_DVC_IN_CODE)
    {
        … 在这里进行所需要的处理动作

        // 因为不返回信息给应用，所以直接返回成功即可
        // 没有用到输出缓冲
        irp->IoStatus.Information = 0;
        irp->IoStatus.Status = STATUS_SUCCESS;
    }
    else
    {
        // 其他的请求不接受，直接返回错误。请注意这里返
        // 回错误和前面返回成功的区别
        irp->IoStatus.Information = 0;
        irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
    }
    IoCompleteRequest (irp,IO_NO_INCREMENT);
```

```
    return irp->IoStatus.Status;
}
```

在前面设置分发函数的时候，要加上：

```
    DriverObject->MajorFunctions[IRP_MJ_DEVICE_CONTROL] =
MyDeviceIoControl;
```

在应用程序方面，进行 DeviceIoControl 的代码如下：

```
    HANDLE device=CreateFile("\\\\.\\MyCDOSL",
        GENERIC_READ|GENERIC_WRITE,0,0,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_SYSTEM,0);
    BOOL ret;
    DWORD length = 0;          // 返回的长度

if (device == INVALID_HANDLE_VALUE)
{
    // … 打开失败，说明驱动没加载，报错即可
}

    BOOL ret = DeviceIoControl(device,
        MY_DVC_IN_CODE,        // 功能号
        in_buffer,             // 输入缓冲，要传递的信息，预先填好
        in_buffer_len,         // 输入缓冲长度
        NULL,                  // 没有输出缓冲
        0,                     // 输出缓冲的长度为 0
        &length,               // 返回的长度
        NULL);

if(!ret)
{
    // … DeviceIoControl 失败。报错
}

// 关闭
CloseHandle(device);
```

7.2.4　驱动层信息传出

驱动主动通知应用和应用通知驱动的通道是同一个，只是方向反过来。应用程序需要开启一个线程调用 DeviceIoControl（调用 ReadFile 亦可）；而驱动在没有消息的时候，则阻塞这个 IRP 的处理，等待有消息的时候返回。

有的读者可能听说过在应用层生成一个事件，然后把事件传递给驱动。驱动有消息要通知应用的时候，则设置这个事件。但是实际上这种方法和上述方法本质相同：应用都必须开启一个线程去等待（等待事件），而且这样使应用和驱动之间交互变得复杂（需要传递事件句柄）。这毫无必要。

让应用程序简单地调用 DeviceIoControl 就可以了。当没有消息的时候，这个调用不返回，应用程序自动等待（相当于等待事件）；有消息的时候这个函数返回，并从缓冲区中读到消息。

实际上，驱动内部要实现这个功能，还是要用事件的，只是不用在应用和驱动之间传递事件了。

驱动内部需要制作一个链表，当有消息要通知应用的时候，则把消息放入链表中（请参考前面的“使用 LIST_ENTRY”一节），并设置事件（请参考前面的“使用同步事件”一节），在 DeviceIoControl 的处理中等待事件。下面是一个例子，展示的是驱动中处理 DeviceIoControl 的控制码为 MY_DVC_OUT_CODE 的部分。实际上，驱动如果有消息要通知应用，必须把消息放入队列尾并设置事件 g_my_notify_event。MyGetPendingHead 获得第一条消息。请读者用以前的知识自己完成其他的部分。

```
NTSTATUS MyDeviceIoCtrlOut(PIRP irp,ULONG out_len)
{
    MY_NODE *node;
    ULONG pack_len;
    //:获得输出缓冲区
    PVOID buffer = irp->AssociatedIrp.SystemBuffer;

    // 从队列中取得第一个。如果为空，则等待直到不为空
    while((node = MyGetPendingHead()) == NULL)
    {
        KeWaitForSingleObject(
            &g_my_notify_event,// 一个用来通知有请求的事件
            Executive,KernelMode,FALSE,0);
    }

    // 有请求了，此时请求是 node，获得 PACK 要多长
    pack_len = MyGetPackLen(node);
    if(out_len < pack_len)
    {
        irp->IoStatus.Information = pack_len;// 这里写需要的长度
        irp->IoStatus.Status = STATUS_INVALID_BUFFER_SIZE;
            IoCompleteRequest (irp,IO_NO_INCREMENT);
```

```
        return irp->IoStatus.Status;
    }

    // 长度足够，填写输出缓冲区
    MyWritePackContent(node,buffer);
    // 头节点被发送出去了，可以删除了
    MyPendingHeadRemove ();
        // 返回成功
    irp->IoStatus.Information = pack_len;// 这里写填写的长度
    irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

这个函数的处理要追加到 MyDeviceIoControl 中，如下所示：

```
NTSTATUS MyDeviceIoControl(
    PDEVICE_OBJECT   dev,
    PIRP   irp)
{
    …
    if(code == MY_DVC_OUT_CODE)
        return MyDeviceIoCtrlOut(dev,irp);
    …
}
```

在这种情况下，应用可以循环调用 DeviceIoControl，来取得驱动通知它的信息。

探索篇

研究内核

本书的第三部分，开始探索 Windows 内核程序，并尝试阅读反汇编代码作为指引。本部分包括第 8~10 章。如果读者对 Windows 内核编程已经有一定的了解，这一部分会比较有趣；如果读者从未接触过 Windows 内核编程，读者应该先学习第二部分。能自己编写内核程序并不意味着可以读懂内核，虽然反过来是一定成立的。读懂别人编写的没有代码的程序，比自己编写更困难一些，但的确是值得的。

第 8 章　进入 Windows 内核

8.1　开始 Windows 内核编程

8.1.1　内核编程的环境准备

基础知识

现在是把我们在前面学到的内核编程知识用于实际的时候了。

内核程序和驱动程序的概念在本书中完全视为等价，它们大部分被编译成.sys 文件放置在 Windows\System32\Drivers 目录下。这些代码将运行在 R0 层，拥有高权限。读者在这里将开始 Windows 内核编程，一些读者可能已经习惯使用 VC 开发 Windows 应用程序，但是从未开发过内核程序。

仅仅安装 Visual Studio 不足以进行 Windows 内核开发，还需要下载微软的驱动程序开发包。在以前，这个开发包叫做 DDK（Driver Development Kit），从大约 2006 年开始，这个开发包被升级为 WDK（Windows Driver Kit），并且可以在微软的网站 http://connect.microsoft.com/上免费下载，下载前需要先在网站上注册。读者也可以在网上搜索其他下载地址。

上机实践

WDK 这个包很大，解压后超过 1GB，读者需要耐心下载和安装。

WDK 和 Visual Studio 之间没有什么关系，可以安装到任何目录。安装的时候请选择完全安装，否则可能被跳过一些有用的例子。

安装成功之后，读者就可以开始建立第一个 Windows 内核程序了。注意：这和以前开发应用程序不同，并不是在 Visual Studio 中建立一个工程，增加各种文件，然后单击 build 菜单这种操作方式。

WDK 安装好之后，在“开始”→“所有程序”菜单中出现“Windows Driver Kits”子菜单，打开，选择“Build Environments”，下面出现了各个不同版本的操作系统，包括从 2000~2003、Vista，每种操作系统下又有不同的选择。

在很多情况下，Windows 的不同版本之间的内核程序是二进制兼容的。也就是说，同一个 sys 文件放到 2000 下和 2003 下可能都能运行。但是，由于 Windows 内核的不断升级，内核调用个数的增加，会导致有时不兼容的情况出现。因此，在编译内核程序的时候，应该选定一个版本的目标操作系统。

此外，常见的选项还有平台选项：x86、x64、IA64。x86 指的就是 IA32 平台，我们的计算机最常用的（只要用的 Windows 是 32 位版本，和 CPU 是 32 位还是 64 位无关），x64 指的是 AMD64，如果在 PC 上使用 64 位版本的 Windows XP、2003 或者 Vista，一般都是 AMD64。IA64 是 Intel 的与 x86 不兼容的 64 位平台，很少被用到。

check 和 free 版本意味着“用于调试”的版本和“用于发布”的版本，与应用编程中的“Debug”版和“Release”版的意义是对应的。

那么请选择一个版本来构建环境吧。我这里选择“Windows XP x86 Checked Build Environment”，点击之后出现一个控制台，这个控制台里已经帮我们配置好了所有需要的环境变量。

初次学习的时候，使用 WDK 下面的范例是最简洁的方法，省去自己编写一个工程的麻烦。WDK 一般默认安装在一个叫做 WinDDK 的目录下，我将它安装在 D 盘。现在，打开 D:\winDDK\6000\src\storages\filters 目录，这里有存储设备过滤驱动的例子。

下面有一个目录叫做 diskperf，这是一个简单的磁盘过滤驱动程序。把这个目录拷贝到其他任意地方，然后在前面打开的控制台中（是指点击 WDK 在“开始”菜单中的“Windows XP x86 Checked Build Environment”菜单后弹出的控制台）进入该目录。

```
cd diskperf
build
```

输入 build 之后，出来结果如下：

```
Linking Executable - objchk_wxp_x86\i386\diskperf.sys
BUILD: Finish time: Mon Mar 24 12:34:43 2008
BUILD: Done
```

这说明，diskperf.sys 已经编译到 objchk_wxp_x86\i386\目录下了。驱动生成之后，最需要的就是让它运行起来。

读者当然可以把这个驱动直接安装在自己的计算机上。但是刚刚编写的驱动常常含有 BUG，在加载瞬间蓝屏，导致自己正在编辑的代码文件损坏并不是非常罕见的“事故”。

解决方案：使用专门的测试机或者在计算机中安装一个“虚拟机”来加载驱动进行测试。关于虚拟机的安装，在“8.2 学习用 WinDbg 进行调试”一节中有详细介绍。

下面要让这个驱动程序在虚拟机的 Windows 系统中运行。在 diskperf 的目录下，有一个扩展名为 inf 的文件，把编译出的 sys 文件和这个文件放在同一个目录下，拷贝到虚拟机中。然后在虚拟机中，对 inf 文件单击鼠标右键，选择“Install”（安装）。

重启虚拟机中的 Windows 系统之后，这时 diskperf 就已经在运行了。diskperf 执行磁盘效率统计，但是我们看不到任何现象，必须调试它才能了解到它的存在。在“8.2 学习用 WinDbg 进行调试”一节中将介绍如何使用 WinDbg 进行调试。

所有的代码都在 diskperf 中。读者可能不习惯没有使用 Visual Studio 进行编译，对于这个问题的解决，可以自己建立一个工程，然后把 c 文件和头文件加入到工程里，这样可以方便编辑。但是依然不能编译，必须用前面的控制台 build 命令进行编译。此外，Visual Studio 不能调试内核程序。

也可以建立一个 Makefile 工程，然后把 build 命名做在批处理指令中，交给 Makefile 工程执行，这样就可以使用 Visual Studio 进行编译了。但是这并不是必需的，所以本书不做详细介绍了。

8.1.2　用 C 语言写一个内核程序

上面利用了 WDK 中的例子 diskperf，但这只是一个专门的磁盘过滤驱动程序。Windows 驱动程序有许多的类别，拥有庞大的知识体系，包括网络、硬盘、文件系统、USB、声卡/显卡、各种打印机，各种层次上的驱动程序，有各种不同的接口、函数和知识体系。本书并没有一一介绍各个类别的驱动程序，前面只介绍了驱动开发中通用的编程方法。

希望进一步学习 Windows 驱动程序的读者，我推荐《Programming the Microsoft Windows Driver Model》一书，这本书在网上有流传的电子版下载，而且是经过翻译的中文版。英文版作者为 Walter Oney。

由于本书电子版网上没有官方的链接，所以请读者自己搜索书名寻找下载地址。

使用 C 语言写一个内核程序，当然要建立一个 C 语言源文件。编译的方法在上一节已经介绍了，但是编译的是 diskperf.c。目前并不修改工程配置，而是直接修改 diskperf.c 这个文件。如果读者需要完全独立地制作一个工程，那么请参照 9.1.1 节“建立一个 C++

的内核工程”，本节中详细介绍了 source 文件的修改方法。

diskperf.c 中有很多复杂的代码，暂时读者不需要去读，有兴趣的读者如果以后学习磁盘过滤驱动程序，就会理解那些代码。在本书中，我们只研究不依赖特定框架的独立的内核程序。

为了演示一个最简单的环境，现在请将这个.c 文件的内容清空，然后写下如下的代码：

```
#include <ntifs.h>
NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    return STATUS_SUCCESS;
}
```

习惯编写应用程序，但是从未接触过内核编程的程序员常常会疑惑：一个内核程序是否生成一个进程？或者一个内核程序是否是一个被人调用的 DLL？实际上，一个内核程序可以被看做一个 PE 格式的 DLL，它是被 Windows 整个内核（而不是某个应用程序）调用的一个 DLL，一旦加载，就成为内核的组成部分，并非特定处于某个进程中。所有内核内存空间是共享的。换而言之，如果在一个内核程序中分配了一个内核指针，然后传递给其他的内核程序，这种通信方法是有效的。内核程序一旦崩溃，Windows 系统也就崩溃了，这时一般都会蓝屏或者自动重启。

DriverEntry 是一个内核程序的入口，在内核程序被加载的时候，这个函数被调用（加载时所在进程一般是名为“System”的进程。Windows XP 下这个进程的 PID 为 4，Windows 2000 下这个进程的 PID 为 8），C 源代码中必须提供这个函数。在完全不需要与外界交互的情况下，上面的代码就足够了：这是一个可以被加载的驱动程序。当然，读者可以在 return STATUS_SUCCESS 之前，加入要编写的代码。

编写内核程序的规则如下：

- 不能调用 Windows 应用层 API 函数。
- 许多 C 标准函数失去了意义，比如 printf，因为内核程序并没有控制台，fopen、fread、fwrite 也失去了意义。在内核中操作文件使用相关内核 API。请参考本书第 5 章“文件与注册表操作”。另外，也不能用 connect、select 等来建立网络连接。相关操作涉及网络驱动的开发，这在本书范围之外。

- 许多比较单纯的 C 标准函数，如 string.h 中的字符串处理函数等，还是可以使用的。但是推荐使用 Rtl 系列内核 API。请参考本书第 4 章“内核字符串与内存”。
- 可以用标准的 C 语言，但是不能直接用浮点数。如果要用浮点数，需要特殊处理。
- 不要用任何应用层中的方法来分配和释放内存。相关方法参考本书第 4 章“内核字符串与内存”。
- 在 WDK 的帮助里，有大量的所谓 System Routine 可以调用。我把它们叫做“内核 API”，因为我觉得它们和 Win32 API 很像。

在内核中可以做任何事情，包括访问任何进程空间内的地址（这可能需要一些技巧），或者修改 Windows 内核原有的函数调用，访问各种控制寄存器，通过 IO 操作硬件，可以说无所不能。当然，这是有代价的。草率的、勉强的编程，会导致 Windows 系统的不稳定，所以请一定以慎重的态度编写每一行代码。

对于上面编写的最简单的驱动程序，请不要使用 diskperf 的 inf 文件安装。

在 Windows 中，不同类型设备的驱动有不同的安装方式，这些方式被写成一个脚本，也就是 inf 文件。但是，对于完全“非任何类型”的驱动程序，则可以很简单地当做一个服务安装。

要制作一个这样的驱动程序和制作一个服务程序的安装程序是一样的，都要用 CreateService 等几个 API（注意这是应用层 API）函数。但是与其自己开发一个安装程序，不如用现成的服务安装工具。

请在网络上搜索“服务安装卸载工具”。作者本人使用 Installer.exe。但是请不要搜索“Installer.exe”，因为有太多的软件叫这个名字。

请把该驱动直接当做一个服务进行安装（假设读者已经按照上面的指引找到了服务安装的工具程序）。

安装之后，在控制台输入命令：

```
net start 服务名
```

这个驱动就会被加载，DriverEntry 中的代码就会被执行。现在这个驱动没有加入任何代码，在本书第三部分“动手开发”中，会尝试往里面添加许多代码。

请注意服务名和驱动文件的名称没有关系，服务名是安装服务的时候指定的。同时，

如果这个服务是一个驱动，那么在安装过程中要指定驱动程序的文件名和所在的路径。但是文件名和服务名之间并没有什么必要的关联。

8.2 学习用 WinDbg 进行调试

我并不是 Windows 调试专家，但为了某些工作方面的任务，我需要用到 Windows 的调试工具：WinDbg。这个工具非常重要，功能无比强大，几乎所有的 Windows 内核程序员都使用它调试 Windows 的内核程序。调试 Windows 程序是一种专业而艰深的技术。下面只做基础的讲解，使读者可以马上开始实践，以便后面的学习。

有些读者可能希望进一步学习调试技术，我推荐《软件调试》一书。这本书的信息如下：

书名：软件调试

作者：张银奎

出版社：电子工业出版社

本书详尽地介绍了包括 WinDbg 在内的调试工具，以及软件调试方法。

能进行 Windows 内核调试的软件较少，除了 WinDbg 之外，SoftIce 也可以一用，SoftIce 的好处是可以直接进行单击调试。但是遗憾的是，这个软件现在已经停止开发。此外，还有由国内人士开发的调试工具 Syser。本书推荐使用 WinDbg。

8.2.1 软件的准备

首先必须下载 WinDbg，这个软件由微软免费提供，可以在微软的网站上找到下载链接，也可以通过 google 搜索。本书编写时，这个链接的地址为：

http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx

如果读者发现此地址失效，请在 google 中输入“WinDbg”进行搜索。

WinDbg 的调试方法设计用于双机调试，一台计算机为调试机，而另一台计算机为被调试机，这两台计算机用串口线相连。

有时读者可能没有足够多的计算机，这里推荐使用虚拟机作为被调试机的方法，这样就可以在单机上调试了。

虚拟机是一种安装在 PC 上的软件，安装之后，完全模拟一台虚拟的 PC，读者可以像操作真正的计算机那样操作它，在其中安装不同的操作系统。

虚拟机推荐 VMware，这是一个付费软件，但是读者可以先找到试用版本。VMware 运行后，请先生成一个虚拟机，并在这个虚拟机上安装 Windows。这个操作请参考 VMware 的说明，和在真机上安装 Windows 相差无几，这里不再详述。本书编写过程中，我发现 VMWare 的官方网站正在维护，无法正常下载试用版。因此，下面提供华军软件园的一个下载地址：

http://www.onlinedown.net/soft/2062.htm

很难保证这个下载地址始终有效，因此需要的读者，可以在 google 中输入“VMWare”进行查找。

虚拟机软件有许多种，除了 VMWare 之外，VirtualPC 也是不错的选择。但是下面的操作都以 VMWare 为例。

8.2.2　设置 Windows XP 调试执行

安装好虚拟机之后，必须把这个虚拟机上的 Windows 设置为调试执行。打开虚拟机中 Windows 的系统盘，在文件夹选项中设置为显示所有文件，不隐藏系统保护的文件，然后可以在系统盘下看到一个文件 boot.ini。

这是除了 Windows Vista 之外的情况，关于 Vista 的情况后面再作说明。

boot.ini 的内容一般如下：

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft    Windows    XP
Professional" /noexecute=optin /fastdetect
```

将最后那一行复制一行，并加上新的参数即可：

```
[boot loader]
timeout=30
```

```
    default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
    [operating systems]
    multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP
Professional" /noexecute=optin /fastdetect
    multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP
Professional" /noexecute=optin /fastdetect /debug /debugport=com1
/baudrate=115200
```

保存这个 ini 文件（这个文件若无法保存，请用鼠标右键单击这个文件，选择“属性”，然后去掉“只读”钩），重启系统，在启动的时候就可以看到菜单，可以进入正常的 Windows XP，也可以进入 Debug 模式的 Windows XP。

8.2.3 设置 VMWare 虚拟机调试

进入 Debug 模式的 Windows XP 后可能会很快死机，但不用担心，这是因为没有连上调试器的缘故。现在在虚拟机上指定一个用管道虚拟的串口（这一段必须详细描述，因为在帮助文档中找不到答案）：

用 VMWare 打开所建立的虚拟机，但是不要启动。在左边的 Commands 栏中点击“Edit virtual machine settings”，出现 Hardware 页，单击下面的“Add”，出现 Add Hardware Wizard，单击“下一步”按钮。Hardware types 选择 Serial Port（串口），单击“下一步”按钮，选择“Output to named pipe”。单击“下一步”按钮，有三个框可以选择：前两个保持默认，分别为：“\\.\pipe\com_1”和“This end is the server.”，第三个修改为“The other end is an application.”，然后单击“完成”按钮即可。

虚拟机这样设置就可以了，下面是 WinDbg 的启动参数，必须使用下面的参数启动安装在本机上的 WinDbg（注意是安装在本机上的，而不是虚拟机中的）。

```
windbg.exe -b -k com:port=\\.\pipe\com_1,baud=115200,pipe
```

打开虚拟机，启动到调试模式下的 Windows 之后，马上以这个参数启动 windbg.exe，就可以开始调试了，windbg.exe 会显示连接上的信息。当然每次这样去输入指令参数是不方便的，方便的方法是在桌面上做一个快捷方式，然后把前面的参数加到快捷方式中的“目标”里。

以上的方案最早出自国外的论坛，感谢 jiurl 将它翻译成中文，从而得以在国内被广为使用。

8.2.4　设置被调试机为 Vista 的情况

下面是 Vista 的情况。Vista 已经不再使用 boot.ini 文件，而是在 Vista 启动之后，打开控制台（在“运行”中输入 cmd 并回车），首先输入：

```
bcdedit /?
```

这会显示一段眼花缭乱的帮助文字，不管能看懂多少，请先了解一下。然后我们需要列举出现在在本机上所有的“操作系统加载器”的情况，请输入：

```
bcdedit /enum OSLOADER
```

如果是刚刚安装的 Vista，一般只有一个标识为{current}的 OSLOADER，这就是当前的启动配置。现在需要的是建立一个新的启动配置，从完全空白开始建立非常麻烦，拷贝一个是简单的方案。

```
bcdedit /copy {current} /d "Windows Vista Copy"
```

拷贝之后会提示已经建立了一个新的配置。新的配置的标识非常长，是一个 GUID 字符串。要设置新的配置简直是一个噩梦（每次都要设法输入那个 GUID 字符串），但是没有必要，幸好可以设置当前配置。输入以下两条指令即可：

```
bcdedit /debug ON
bcdedit /bootdebug ON
```

输入结束之后，还可以输入一条指令看一下当前的调试配置：

```
bcdedit /dbgsetting
```

一般来说，会显示出使用的第一个串口，波特率为 115200，和期望的一致，所以不需要修改。最后指定一下选择菜单的超时，太长（30 秒）在无人启动的时候会太慢，太短则在试图选择的时候下手不及。我个人觉得 7 秒非常合适：

```
bcdedit /timeout 7
```

然后重新启动，原来的配置就是调试模式了。如果不进入调试模式，那么选择“Windows Vista Copy”进入即可。WinDbg 和 VMWare 上的配置不需要修改，和调试 WindowsXP 的情况完全相同。

刚刚连接上的时候，虚拟机里的 Windows 系统会被中断，貌似死机，这时，请在 WinDbg 的命令提示符“kd>”后面输入“g”并回车。

8.2.5 设置 Windows 内核符号表

现在虚拟机里的操作系统进入了调试模式，那么怎样调试之前编译的 diskperf 呢？实际上没有必要告诉 WinDbg 需要调试的是 diskperf，WinDbg 把内核视为一个整体，我们只要告诉它代码的路径和符号表的路径就可以了。在调试连接上之后，打开 WinDbg 的主菜单“File”下的“Symbol File Path”，在这里输入符号表位置。符号表和 sys 产生在同一个目录下，所以只要指定本机上编译结果所在的 objchk_wxp_x86\i386 目录就可以了。

如果有多个驱动需要调试，那么可以指定多个路径，路径之间用分号分隔。

此外，需要指定 Windows 的内核符号表。Windows 的每一个 sys 文件都有自己的符号表，这些符号表需要从网上下载，但是没有必要自己去下载。在 Symbol File Path 中增加如下一条设置，用分号与其他路径隔开：

```
srv*c:\ symbols*http://msdl.microsoft.com/download/symbols
```

这条设置使 WinDbg 自动用 HTTP 协议从微软的网站上下载所需要的符号表。首次使用会使 WinDbg 变得极慢，接近死机，但是没有关系，实际上 WinDbg 正在下载符号表，可以打开 c:\symbols 查看，这个路径也可以指定为别的位置。

并不是所有的时候都可以成功下载符号表，我不清楚是因为其他设置还是网络问题，有几次我下载符号表始终不成功，但是大部分时候是成功的。下载完成后执行就很快了。读者可以在调试过程中看见微软提供的所有的函数名，也可以随时查看调用栈。

如果调试的时候总是看不见可懂的函数名，是符号表设置问题。请正确设置参数并保持网络畅通。

8.2.6 调试例子 diskperf

现在来调试 8.1 节中编译的例子程序 diskperf。diskperf 不是 Windows 系统内核原装组件，因此不可能下载得到符号表。但是实际上，符号表会在编译过程中，产生在和 sys 文件同一个目录下。在上面，已经设置了 Symbol File Path。现在在设置好的字符串后加分号，然后加入 diskperf 的符号表路径，diskperf 的符号表路径就是编译出.sys 文件所在的路径，请写全路径。

如果有 C 语言代码，请打开 WinDbg 主菜单“File”下的“Source File Path”。由于

我们没有 Windows 内核的全部代码，这里输入 diskperf 代码的路径，也可以指定多个路径，用分号分隔。

现在选择 WinDbg 主菜单中“Debug”下的“Break”，中断 Windows 执行。此时可以用“File”→“Open”打开 diskperf 下的 C 语言文件，选定一行，按下 F9 键，此行变红，就下了一个断点。

除了 diskpef 之外，还可以调试 Windows 中的任何函数，比如 IoCreateFile。现在输入命令：

```
u IoCreateFile
```

输出结果如下：

```
kd> u IoCreateFile
nt!IoCreateFile:
8056cc9a 8bff    mov     edi,edi
8056cc9c 55      push    ebp
8056cc9d 8bec    mov     ebp,esp
8056cc9f 83ec0c  sub     esp,0Ch
8056cca2 53      push    ebx
8056cca3 56      push    esi
8056cca4 33f6    xor     esi,esi
8056cca6 8975fc  mov     dword ptr [ebp-4],esi
```

这是 IoCreateFile 的开头部分。打开 WinDbg 主菜单“View”→“Disassembly”，在 offset 中输入所看到的 IoCreateFile 开始地址 8056cc9a，马上就可以看到 IoCreateFile 完整的汇编代码了。请自己尝试一下。

现在回到 diskpef 的调试，读者可以在 Disassembly 窗口中选定一行，直接按 F9 键设置断点。同理，打开已经设置好路径的 C 文件，按下 F9 键也可以设置断点。之后的调试过程，与 Vistual Studio 下调试 C 程序完全类似。

8.3　认识内核代码函数调用方式

基础知识

很少有人能直接阅读机器码，要分析必须得到汇编指令。IDA 是静态反汇编的工具，

它帮你将 sys 文件变成汇编代码。网络上有很多文章介绍使用 IDA 的方法，如果遇到困难，建议搜索一下。

IDA 可以在网上下载试用版。这个软件的使用极其简单，没有什么复杂的配置。直接将一个 exe 或者 sys 文件拖入该软件中，一阵反汇编计算之后，就能看见汇编指令、函数导入导出表、字符串表等重要信息。

把 sys 文件用 IDA 进行反汇编时，如果没有符号表，得到的函数和全局变量都没有可理解的名字。但是幸运的是，微软提供的所有 Windows“原装的”接口都能在反汇编代码中看到函数名。

所以后面的反汇编都假设总是可以看到微软的符号表。如果有的 sys 没有符号表，将看不到它内部的函数名，但是总还是可以看到调用的 Windows 系统调用的函数名。

现在开始把前面对 C 语言反汇编的练习投入实用，然而，请暂时不要直接去阅读更复杂的内核代码，那样读者可能会失去信心。所以我从 WDK 中把例子 diskperf 完整地取出编译了一个发行版反汇编了一下，算做初步的练习。这有几个好处，首先，diskperf 仅仅 2000 多行代码，就算最差的情况逐行阅读，精力的损失也在可预测的范围之内；其次，diskperf 机制简单，也不担心过于不明的原理与机制造成困难。C 代码就在你的手中，解读之后，还有机会比较正确答案。

代码分析

拿到 diskperf.sys 文件并用 IDA 进行反汇编之后，首先看入口函数 DriverEntry，相关的反汇编结果如下。现在就来假定这是一个从没见过 C 代码的 Windows 内核组件，来尝试恢复它为 C 语言。

```
INIT:00011480 DriverEntry     proc near
INIT:00011480
INIT:00011480 arg_0  = dword ptr  8
INIT:00011480 arg_4  = dword ptr  0Ch
INIT:00011480

;以下 push 的操作都是 F 指令，目的是保存 esi、edi。然后 mov
;是取参数 arg_0。arg_4 是 IDA 自己定义的常数，用来取得参
;数。[esp+arg_4]就是[esp+0Ch]，是第二个外部参数
INIT:00011480    push     esi
INIT:00011481    mov      esi, [esp+arg_4]
```

```
INIT:00011485    mov     ax, [esi]
INIT:00011488    add     ax, 2
INIT:0001148C    push    edi
INIT:0001148D    mov     DiskPerfRegistryPath.MaximumLength, ax
INIT:00011493    movzx   eax, ax

;下面是调用 ExAllocatePoolWithTag 分配内存的过程，IDA 把
;参数都标示了
INIT:00011496    push    66725044h    ; Tag
INIT:0001149B    push    eax          ; NumberOfBytes
INIT:0001149C    push    1            ; PoolType
INIT:0001149E    call    ds:__imp__ExAllocatePoolWithTag@12
INIT:000114A4    test    eax, eax
INIT:000114A6    mov     DiskPerfRegistryPath.Buffer, eax
INIT:000114AB    jz      short loc_114BB

;以上的 jz 显然涉及一个 if。只有 eax 不为 0 才执行下面的指令
;否则跳到 114BB 去了。下面进行一个字符串拷贝
INIT:000114AD    push    esi
INIT:000114AE    push    offset DiskPerfRegistryPath
INIT:000114B3    call    ds:__imp__RtlCopyUnicodeString@8
INIT:000114B9    jmp     short loc_114CB

;以上这个跳转依然是上面的 if 的一部分，这避免了执行下面的
;给 DiskPerfRegistryPath.Length
;和 DiskPerfRegistryPath.MaximumLength 赋 0 的操作
INIT:000114BB
INIT:000114BB loc_114BB:
INIT:000114BB    and     DiskPerfRegistryPath.Length, 0
INIT:000114C3    and     DiskPerfRegistryPath.MaximumLength, 0
INIT:000114CB

; test-jz，然后 jmp，与前面所说的 if 为 cmp-jl，然后后面的 else 只有一
;个 jmp 是一个意思。说明上面是一个单 if-else 结构，没有 else if 出现
INIT:000114CB loc_114CB:

;下面取了第 1 个参数放到了 edx 中。若 edx+38h 取内容，则第一个
;参数要么是数组，要么是结构体。当然，因为这个函数是 DriverEntry，
;我们很清楚第一个参数类型是 DRIVER_OBJECT，也容易查到
;+38h 的位置是什么域
INIT:000114CB    mov     edx, [esp+4+arg_0]
INIT:000114CF    lea     esi, [edx+38h]
```

```
;以下指令等于 mov ecx,1Ch，这看起来是一个循环的开始（ecx 常
;常用来做循环变量）
INIT:000114D2    push    1Ch
INIT:000114D4    pop     ecx
INIT:000114D5    mov     eax, offset DiskPerfSendToNextDriver
INIT:000114DA    mov     edi, esi
INIT:000114DC    rep stosd

;下面是漫长的赋值操作，最后是返回
INIT:000114DE    mov     eax, offset DiskPerfReadWrite
INIT:000114E3    mov     [edx+44h], eax
INIT:000114E6    mov     [edx+48h], eax
INIT:000114E9    mov     eax, offset DiskPerfShutdownFlush
INIT:000114EE    mov     [edx+78h], eax
INIT:000114F1    mov     [edx+5Ch], eax
INIT:000114F4    mov     eax, [edx+18h]
INIT:000114F7    mov     dword ptr [esi], offset DiskPerfCreate
INIT:000114FD    mov     dword ptr [edx+70h],
                         offset DiskPerfDeviceControl
INIT:00011504    mov     dword ptr [edx+94h], offset DiskPerfWmi
INIT:0001150E    mov     dword ptr [edx+0A4h],
                     offset DiskPerfDispatchPnp
INIT:00011518    mov     dword ptr [edx+90h],
                     offset DiskPerfDispatchPower
INIT:00011522    mov     dword ptr [eax+4],
                     offset DiskPerfAddDevice
INIT:00011529    pop     edi
INIT:0001152A    mov     dword ptr [edx+34h],
                     offset DiskPerfUnload
INIT:00011531    xor     eax, eax
INIT:00011533    pop     esi
INIT:00011534    retn    8        ;retn 8，返回的同时恢复堆栈
INIT:00011534 DriverEntry    endp
```

以上函数调用方式和前面所见的标准 C 函数调用方式的区别是：由被调用函数自己恢复堆栈。retn 8 等于调用 ret 后，再把 esp+8。这是 std call 的函数调用方式，也就是默认的内核函数的调用方法。

8.4　尝试反写 C 内核代码

代码分析

要看懂内核代码，前提是应该会使用 C 语言开发内核驱动程序，并充分利用 WDK 中的范例。既然微软已经为 WDK 提供了详细的文档和例子，就应该好好利用。

下面打开 WDK（或者 DDK）的 Help 文档，这些文档的快捷方式在“开始”菜单中。你将不断看到内核数据结构和函数调用，当碰到没有见过的调用和结构的时候，请在 Help 文档中查找它们加以熟悉。除了稍加解说，本书不会罗列帮助中的内容来占用篇幅。

现在尝试用前面学到的方法来反写 C 代码。diskperf 的代码手边就有，但是如果对照着来写，这个练习也就没有意义了，请假装没看过 C 代码。先看开头的一段：

```
INIT:00011480    push    esi                     F
INIT:00011481    mov     esi, [esp+arg_4]
INIT:00011485    mov     ax, [esi]
INIT:00011488    add     ax, 2
INIT:0001148C    push    edi                     F
INIT:0001148D    mov     DiskPerfRegistryPath..MaximumLength, ax
INIT:00011493    movzx   eax, ax
```

首先忽略两条 F 指令，剩余的就简单了。mov esi, [esp+arg_4]意味着取第二个参数到esi 中。DriverEntry 的第二个参数是 PUNICODE_STRING RegisterPath，UNICODE_STRING 的第一个域应该是 Length，那么[esi]就是取它的长度了。下面的指令是把这个长度加 2，并保存到全局变量 DiskPerfRegistryPath.MaximumLength 中。

```
NTSTATUS DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegisterPath)
{
    DiskPerfRegisteryPath.MaximumLength = RegisterPath.Length + 2;
    … …
}
```

然后是分配内存的代码：

```
INIT:00011496    push    66725044h    ; Tag
```

```
INIT:0001149B    push    eax             ; NumberOfBytes
INIT:0001149C    push    1               ; PoolType
INIT:0001149E    call    ds:__imp__ExAllocatePoolWithTag@12
INIT:000114A4    test    eax, eax
INIT:000114A6    mov     DiskPerfRegistryPath.Buffer, eax
```

请回忆前面的知识。ExAllocatePoolWithTag 被调用之后，这个函数的返回结果放在 eax 中。因此，后面紧接着就检查 eax 并传递到 DiskPerfRegistryPath.Buffer 中。

以上这个函数调用过程，几乎简单得不用翻译。但是汇编中把返回值要 mov 来 mov 去的，C 语言毕竟是高级语言，就不要那么笨拙了。

```
DiskPerfRegistryPathBuffer.Buffer = ExAllocatePoolWithTag(1,
 DiskPerfRegisteryPathMaximumLength,0x66725044);
```

接着是根据对分配内存结果的判断，当 eax 不为 0 的时候，则拷贝字符串（jz short loc_114BB）。判断语句就是上面的 test eax,eax。

```
INIT:000114AB    jz      short loc_114BB
INIT:000114AD    push    esi
INIT:000114AE    push    offset  DiskPerfRegistryPath
INIT:000114B3    call    ds:__imp__RtlCopyUnicodeString@8
INIT:000114B9    jmp     short loc_114CB
INIT:000114BB
INIT:000114BB loc_114BB:
INIT:000114BB    and     DiskPerfRegistryPath.Length, 0
INIT:000114C3    and     DiskPerfRegistryPath.MaximumLength, 0
INIT:000114CB
INIT:000114CB loc_114CB:
```

那么以上的 if-else 模块大致如下：

```
if(DiskPerfRegistryPathBuffer.Buffer == NULL)
{
    DiskPerfRegisterPath.Length = 0;
    DiskPerfRegisterPath.MaximumLength = 0;
}
else
{
    RtlCopyUnicodeString(&DiskPerfRegistryPath,RegisterPath);
}
```

再往下是这么一段：

```
;edx = DriverObject
INIT:000114CB    mov     edx, [esp+4+arg_0]
```

```
;esi = DriverObject->DispatchFunctions
INIT:000114CF    lea     esi, [edx+38h]
INIT:000114D2    push    1Ch                     ;ecx = 1Ch
INIT:000114D4    pop     ecx

;这是拷贝的源
INIT:000114D5    mov     eax, offset DiskPerfSendToNextDriver
INIT:000114DA    mov     edi, esi                ;拷贝的目的

;重复拷贝 1Ch 填满从 esi 开始的区域
INIT:000114DC    rep stosd
```

理解 edx+38h 需要对 DriverObject 结构有所了解，但这不是问题，有头文件可以查；一些困难的结构可以通过 WinDbg 的某些插件来了解；还有更多没有公开的结构只能去找找别人的资料碰碰运气，不过完全无法找到资料的话，可以自己假定一个。不论是否理解，都可以把它翻译成 C 语言。

```
for(i=0;i<0x1c;i++)
{
    DriverObject->MajorFunctions[i] = DiskPerfSendToNextDriver;
}
```

后面是类似的填写分发函数，这里就不重复这个过程了。这是第一步尝试，看起来以前对 C 语言反汇编结果阅读的基本方法是都可以用上的。

8.5　如何在代码中寻找需要的信息

代码分析

如果我们给在 DriverEntry 中出现过的各个函数，依次反写为 C 语言，同时把这个操作递归下去，确实可以把整个代码变成 C 源代码。不过做做练习还好，如果出于实际的需求这样做的话，就显得笨拙了。作为内核代码的阅读者，应该迅速地找到自己所需要的信息才对。

diskperf 这个工程大家都是如此的熟悉，以至于难以想出什么让人感兴趣的技术点需要通过反汇编去了解，但是出于练习的需要，这里就假定：读者都了解 diskperf 通过绑定物理磁盘设备来进行磁盘操作的过滤，但是不知道，它是如何发现这些物理磁盘设

备（或者枚举）的？又是如何绑定的呢？是如何和系统磁盘的增加/减少保持同步的呢？现在，读者只有 diskperf 的 sys 文件和符号表，那么就自己来了解这些信息吧。

首先猜测可能的实现方法，然后搜索相关的内核 API 查看其调用方法，这是挖掘内核程序知识的捷径。

很容易想到绑定一个设备会调用 IoAttachDeviceToDeviceStack。打开 IDA 反汇编 diskperf.sys 后，寻找名字 IoAttachDeviceToDeviceStack，只有一处调用：在函数 DiskPerfAddDevice 中。

```
…
PAGE:0001127E loc_1127E:
PAGE:0001127E    mov     eax, [ebp+TargetDevice]
PAGE:00011281    push    eax             ; TargetDevice
PAGE:00011282    mov     [esi+8], eax
PAGE:00011285    push    [ebp+IoObject]  ; SourceDevice
PAGE:00011288    calldS:__imp__IoAttachDeviceToDeviceStack@8
```

IoAttachDeviceToDevice 的第二个参数是 TargetDevice，那么被 push 到堆栈的第一个参数就是 TargetDevice。而这个参数从堆栈中取得，也就是[ebp+TargeDevice]，这里的 TargeDevice 是 IDA 定义的常数。在有符号表的情况下，这些参数都有了有意义的名字，而实际的数值可以在 DiskPerfAddDevice 的开头看到：

```
PAGE:000111E2 DiskPerfAddDevice proc near
PAGE:000111E2
PAGE:000111E2 IoObject       = dword ptr  8
PAGE:000111E2 TargetDevice   = dword ptr  0Ch
```

可见 TargetDevice 的实际数值是 0Ch，这是 DiskPerfAddDevice 传入的第二个参数，而第一个参数名字是 IoObject。

有兴趣的话，现在来反写 C 这个函数的其他部分，以便得到更清晰的解答。下面是函数最开始的部分：

```
PAGE:000111E2    push    ebp
PAGE:000111E3    mov     ebp, esp
PAGE:000111E5    push    ebx
PAGE:000111E6    lea     eax, [ebp+IoObject]
PAGE:000111E9    push    eax             ; DeviceObject
PAGE:000111EA    xor     ebx, ebx
PAGE:000111EC    push    ebx             ; Exclusive
PAGE:000111ED    mov     eax, 100h
```

```
PAGE:000111F2    push    eax             ; DeviceCharacteristics
PAGE:000111F3    push    7               ; DeviceType
PAGE:000111F5    push    ebx             ; DeviceName
PAGE:000111F6    push    eax             ; DeviceExtensionSize
PAGE:000111F7    push    [ebp+IoObject]; DriverObject
PAGE:000111FA    call    ds:__imp__IoCreateDevice@28
```

这一段代码把大量的参数 push 到堆栈里，然后调用函数 IoCreateDevice,这是 1.3 节“C 函数的参数传递”中讲述的内容，如果有些忘记了，注意回头复习一下。注意参数是倒序压入堆栈的。目前知道函数有返回值，但是不知道其类型。为了简单起见，返回 ULONG，同时对于第一个参数 IoObject 本不了解是什么类型，但是 IDA 提示这是一个 DRIVER_OBJECT。

```
ULONG DiskPerfAddDevice(
    PDRIVER_OBJECT IoObject,
    PDEVICE_OBJECT TargetDevice)
{
    NTSTATUS status;
    // 把我们 push 过的参数填入。但是，要除去其中的函数开始时
    // 对 ebp 和 ebx 进行备份的指令
    status = IoCreateDevice(
        IoObject,0x100,0,7,0x100,0,
        (PDRIVER_OBJECT *)&IoObject);
}
```

上面的代码是不是有点荒谬？IoCreateDevice 最后一个参数明明应该是一个 DEVICE_OBJECT，为何把 IoObject 给传了进去？

正确的做法应该是这样的：

```
ULONG DiskPerfAddDevice(
    PDRIVER_OBJECT IoObject,
    PDEVICE_OBJECT TargetDevice)
{
    NTSTATUS status;
    PDEVICE_OBJECT Device;
    status = IoCreateDevice(IoObject,0x100,0,7,0x100,0,&Device);
}
```

是的，但是其实前面的写法也没有错，不要忘记了汇编语言是没有类型概念的。IoObject 在堆栈中传入，函数调用完后被恢复，所以对于这些空间，这个函数可以随意使用，不会对外界造成影响。那么，再在函数内部定义一个：

```
PDEVICE_OBJECT Device;
```

还不如直接利用传入的参数 PDRIVER_OBJECT IoObject，它们虽然类型不同，大小却是一样的，都是 4 个字节的指针而已。这是编译器优化的结果。

发行版的优化代码，往往会充分利用函数中已经使用过的参数空间，来做其他的用途，绝不可以固定的意义来理解每个参数与内部变量。

你看到的答案就是，DiskPerfAddDevice 负责生成设备并绑定原始的设备，而原始设备的来源是 DiskPerfAddDevice 的参数。这个函数被设置在 DriverEntry 中：

```
INIT:000114F4    mov     eax, [edx+18h]
INIT:000114F7    mov     dword ptr [esi], offset DiskPerfCreate
INIT:000114FD    mov     dword ptr [edx+70h],
                         offset DiskPerfDeviceControl
INIT:00011504    mov     dword ptr [edx+94h],
                         offset DiskPerfWmi
INIT:0001150E    mov     dword ptr [edx+0A4h],
                         offset DiskPerfDispatchPnp
INIT:00011518    mov     dword ptr [edx+90h],
                         offset DiskPerfDispatchPower
INIT:00011522    mov     dword ptr [eax+4],
                         offset DiskPerfAddDevice
```

这里的 edx 的位置是 DRIVER_OBJECT,DRIVER_OBJECT 偏移 0x18 之后，取得其地址，就是 DriverExtension；然后 DriverExtension 取第 5 个字节开始的位置，就是 DriverExtension->AddDevice。

第9章　用C++编写的内核程序

9.1 用 C++开发内核程序

9.1.1 建立一个 C++的内核工程

基础知识

你会发现大部分已有的 Windows 内核用 C 语言写成，不过也许会在反汇编 Windows 内核的时候发现越来越多的用 C++编写的代码。实际上，用 C++编写内核程序是完全可以的，国内许多安全软件商喜欢使用 C++开发软件，这样一方面能利用 C++大量的特性，另一方面符合了很多一开始就使用 C++进行开发的程序员的习惯。

Windows 内核使用 C 语言开发可能源于 Windows 开发者的传统。C 语言编译出的代码更加干净、易懂，这样在出现 Bug 的时候调试显得更加方便。但是 C++语言显然提供了更多的特性，更多的减少工程量的方法。在逐渐被熟悉之后，使用 C++的内核程序应该会越来越多。

上机实践

用 C++开发内核程序并没有太多特殊的不同之处，只有少数几个要注意的问题点。下面我们参照前面提到过的 diskperf 来建立一个工程。增加一个文件，这个文件的扩展名应该是.cpp，这样会被编译器以 C++的方式编译，比如文件名为 myproject.cpp。在 source 文件中（读者可以将工程 diskperf 中的 source、makefile 这两个文件拷贝到新建立的工程中）。makefile 这个文件不需要任何修改。source 文件的内容如下，在后面会做一点修改。

```
TARGETNAME=diskperf
TARGETPATH=obj
TARGETTYPE=DRIVER
SOURCES=diskperf.c   \
        diskperf.rc
```

现在我们开始建立自己的工程了。首先 TARGETNAME 应该改为我们自己的工程名字，比如 myproject。其次就是 SOURCES，SOURCES 指定每个要编译的源文件。不过要注意，不要把头文件加进去，头文件始终是包含在 c 文件和 cpp 文件中编译的，如

果放在这里，会被试图独立编译，但是这是错误的。现在我们建立的工程要使用 cpp，所以可以直接加一个 cpp 文件。那么上面的内容改成下面这个样子：

```
TARGETNAME=myproject
TARGETPATH=obj
TARGETTYPE=DRIVER
SOURCES=myproject.cpp
```

下面开始写 myproject.cpp。一个内核程序至少要有一个入口 DriverEntry，那么一个最简单的驱动程序的 myproject.cpp 文件应该有以下内容：

```
#include <ntifs.h>
NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver_object,
    PUNICODE_STRING reg_path)
{
    return STATUS_UNSUCCESSFUL;
}
```

以上是一个不能被加载的驱动程序：加载的时候总是返回不成功。因为它没有实现任何分发函数，导致它无法被成功地加载。遗憾的是，初次编译出现了以下的错误：

```
1>1>d:\winddk\3790\inc\ifs\wnet\ntifs.h(5036) : error C2220: warning
treated as error - no object file generated
1>1>d:\winddk\3790\inc\ifs\wnet\ntifs.h(5036) : error C4162:
'_ReturnAddress' : no function with C linkage found
```

9.1.2　使用 C 接口标准声明

上面的问题和 ntifs.h 的声明完全没有考虑可能被 C++文件引用有关。C++的函数和 C 函数即使是在完全同样的写法的情况下，生成的函数接口也并不相同，这样就需要自己为它考虑一下。将 ntifs.h 内的声明全部确认为 C 声明即可：

```
extern "C" {
    #include <ntifs.h>
}
```

此外，DriverEntry 函数前也应该加上 extern "C"。

在 C++中使用 C 编写的接口，请在前面加 extern "C"。

如果想生成一个类，这个类叫做 MyDriver，打算把分发函数作为虚函数以便于继

承，这不能不说是一个好主意。

```
class MyDriver
{
public:
    MyDriver(PDRIVER_OBJECT driver);
    virtual NTSTATUS OnDispatch(PDEVICE_OBJECT dev, PIRP irp);
private:
    PDRIVER_OBJECT d_driver;
};
```

那么构造函数应该填写 driver 的分发函数指针：

```
MyDriver::MyDriver(PDRIVER_OBJECT driver) : d_driver(driver)
{
    size_t i;
    for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++) {
        driver->MajorFunction[i] = OnDispatch;
    }
}
```

不过遗憾的是，这样会出现编译错误：

```
    error C2440: '=' : cannot convert from 'NTSTATUS (__cdecl MyDriver::*)
(PDEVICE_OBJECT,PIRP)' to 'PDRIVER_DISPATCH'
```

9.1.3 使用类静态成员函数

使用 C++的时候总是要注意，把一个类的一个成员函数当做普通的 C 函数使用是不对的。至少类的成员函数都知道 this 指针，而 C 函数就不知道，所以它们的接口实际上是不一样的。这时候，要实现我们的想法，只能这样做：在类中再声明一个静态函数来作为 Dispatch Function。静态函数可以作为普通的 C 函数使用，但是静态函数无法调用 OnDispatch 这个函数，因为静态函数没有 this 指针，不能调用非静态成员。考虑到 Driver 这种对象在驱动中实际上是一个 singleton，我们可以再定义一个类型为 MyDriver* 的静态成员，这等于一个全局公共的 this 指针，只有 singleton 才可以这样做。以下是声明代码：

```
class MyDriver
{
public:
    MyDriver(PDRIVER_OBJECT driver);
    // 可以派生的 Dispatch 虚函数，目前的实现很简单
    // 直接返回失败
```

```
    virtual NTSTATUS OnDispatch(PDEVICE_OBJECT dev, PIRP irp)
    {  return STATUS_UNSUCCESSFUL;};
    // 静态成员，总是记录唯一被实例化的 MyDriver 指针
    static MyDriver *d_my_driver;
private:
    // 静态成员函数。用来作为 Dispatch 函数使用
    static NTSTATUS sDispatch(PDEVICE_OBJECT dev, PIRP irp);
    PDRIVER_OBJECT d_driver;
};
```

以下是实现代码：

```
// 请注意下面只用 sDispatch 这个静态函数来作为分发函数指针
MyDriver::MyDriver(PDRIVER_OBJECT driver) : d_driver(driver)
{
    size_t i;
    for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++) {
        driver->MajorFunction[i] = sDispatch;
    }
    d_my_driver = this;
}

// 静态的分发函数的实现：调用虚函数，以便以后可以派生
NTSTATUS MyDriver::sDispatch(PDEVICE_OBJECT dev, PIRP irp)
{
    return d_my_driver->OnDispatch(dev,irp);
}

MyDriver *MyDriver::d_my_driver = NULL;
```

有了上面的代码，就可以写 DriverEntry 了。

```
extern "C" NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    MyDriver *MyDriver::d_my_driver = new MyDriver(driver);
    return STATUS_UNSUCCESSFUL;
}
```

9.1.4　实现 new 操作符

不过可能再次让读者失望的是，这个程序还是无法通过链接，错误如下：

```
error LNK2019: unresolved external symbol "void * __cdecl operator
```

```
new(unsigned int)" (??2@YAPAXI@Z) referenced in function "long __stdcall
DriverEntry(struct _DRIVER_OBJECT *,struct _UNICODE_STRING *)" (?DriverEntry
@@YGJPAU_DRIVER_OBJECT@@PAU_UNICODE_STRING@@@Z)
```

很遗憾，因为使用了 new 操作符。但是在驱动开发中，内存分配需要自己来定义，那么自己实现一个吧。

```
void * __cdecl operator new(unsigned int size)
{
    void *pt = ExAllocatePool(NonPagedPool,size);
    if(pt != NULL)
        memset(pt,0,size);
    return pt;
}
```

C++编写驱动很重要的一点是要自己实现 new 操作符。

再次链接，显示成功了。有趣的是，上面实现了 new 操作符，却没有实现 delete 操作符，而程序却可以正常运作，这是因为根本就没有去 delete。请练习实现 delete 操作符，并用 C++实现一个简单的驱动程序。

9.2 开始阅读一个反汇编的类

有人说 C++的出现，是汇编阅读的噩梦。这是因为 C++编译的结果，在熟悉 C 语言编译结果的读者看来，非常的诡异。但是在熟悉 C 语言之前，C 语言编译结果的汇编阅读，又何尝不是噩梦呢。同理，C++也绝对不神秘，只要读者熟悉了 C++编译的特点，就可以像了解 C 语言一样了解它。

9.2.1 new 操作符的实现

本书坚持先了解简单，再研究复杂的观点，所以编译一个 32 位有符号表的调试版本的驱动来看看情况。先看这个 operator new 变成什么了。

```
.text:00011000 ; int __cdecl operator_new(SIZE_T NumberOfBytes)
.text:00011000 operator_new      proc near
.text:00011000
.text:00011000 NumberOfBytes   = dword ptr  8
```

```
.text:00011000
.text:00011000        push    ebp
.text:00011001        mov     ebp, esp
.text:00011003        push    20736649h       ; Tag
.text:00011008        mov     eax, [ebp+NumberOfBytes]
.text:0001100B        push    eax             ; NumberOfBytes
.text:0001100C        push    1               ; PoolType
.text:0001100E        call    ds:ExAllocatePoolWithTag(x,x,x)
.text:00011014        pop     ebp
.text:00011015        retn
.text:00011015
.text:00011015 operator_new     endp
.text:00011015
```

非常简单，和普通的 C 函数没有什么区别，只是函数名实际上并不是 new，而是 operator_new。其中调用的是 ExAllocatePoolWithTag，实际上代码中是 ExAllocatePool，可能是因为 ExAllocatePool 这个函数已经被停止使用，被宏替换掉了。

我在实现 new 操作符的时候，曾经想 new 除了分配内存之外，还应该调用类的构造函数。会不会需要我在实现 new 的时候去调用类的构造函数呢？但是显然无法调用，因为我根本不知道是什么类在初始化。这个疑惑的解决，可以看看在 DriverEntry 中，调用 new 之后的代码：

```
.text:000110C0 DriverEntry    proc near
.text:000110C0 var_8 = dword ptr -8
.text:000110C0 var_4 = dword ptr -4
.text:000110C0 arg_0 = dword ptr  8
.text:000110C0

;下面都是函数入口通常动作。首先入栈 ebp，然后用 ebp 保存 esp
;再分配 8 字节（两个 32 位内部变量空间）
.text:000110C0   push    ebp
.text:000110C1   mov     ebp, esp
.text:000110C3   sub     esp, 8

;现在调用 new。调用的参数为 8，是直接 push，然后再调用的。因为
;这是 cdcel 方式调用，所以堆栈必须由调用者恢复。下面用
;add esp,4 来恢复堆栈
.text:000110C6   push    8    ; NumberOfBytes
.text:000110C8   call    operator_new
.text:000110C8
.text:000110CD   add     esp, 4
```

```
;eax 是返回值，放入内部变量中。判断是否为 0，决定是否跳过后
;面调用构造函数的部分。换句话说，如果 new 返回了 NULL，那么
;就不调用构造函数了；反之则调用，这是 C++编译器生成的指令
;并不是我们自己的设计。
.text:000110D0   mov      [ebp+var_4], eax
.text:000110D3   cmp      [ebp+var_4], 0
.text:000110D7   jz       short loc_110EA
.text:000110D7

;构造函数的参数是第 0 个参数，也就是 PDRIVER_OBJECT。另外
;构造函数的名字是 MyDriver__MyDriver，其他成员函数的命名规则
;也是如此，用双下画线代替双冒号。请注意类对象 new 出来之
;后，地址放入 ecx 中，再调用 MyDriver__MyDriver
.text:000110D9   mov      eax, [ebp+arg_0]
.text:000110DC   push     eax
.text:000110DD   mov      ecx, [ebp+var_4]
.text:000110E0   callMyDriver__MyDriver
.text:000110E0
.text:000110E5   mov      [ebp+var_8], eax
.text:000110E8   jmp      short loc_110F1
.text:000110E8
```

9.2.2 构造函数的实现

下面看看构造函数是如何实现的。我曾经有一个疑惑：在 MyDriver 的构造函数中，显然是知道 this 指针的，但是 MyDriver__MyDriver 的反汇编结果如下，参数里却没有 this。那么 MyDriver__MyDriver 这个函数是怎么知道 this 指针的呢？回头看看 MyDriver_MyDriver 在调用之前，new 出来的指针地址放入了 ecx 中，这才真相大白，系统用了 ecx 专门来传递 this 指针。这是__thiscall 调用方式，是我们以前没有遇到过的。

在 C++编写的类成员函数中，用 ecx 传递 this 指针。

```
.text:00011020 MyDriver__MyDriver proc near
.text:00011020

;实际上，我在这个函数中没有定义任何内部变量，但是这里却出现了
;两个内部变量，这是 C++编译器生成的
.text:00011020 var_8 = dword ptr -8
.text:00011020 var_4 = dword ptr -4
.text:00011020 arg_0 = dword ptr  8
.text:00011020
```

```
;开始阶段和以前了解的函数相同。备份 ebp，然后当前 esp 的值写入 ebp
;在堆栈中腾出两个双字的局部变量
.text:00011020        push    ebp
.text:00011021        mov     ebp, esp
.text:00011023        sub     esp, 8

;ecx 入 var_8，再入 eax，等于是 eax 和 var_8 都变成了 this 指针
.text:00011026        mov     [ebp+var_8], ecx
.text:00011029        mov     eax, [ebp+var_8]

;构造函数做的第一件有实质意义的事情是：在对象的开头部分填入虚
;函数表。换句话说，对象的内存空间一开始就是虚函数表
.text:0001102C   mov dword ptr [eax],
                              offset const MyDriver::`vftable'

;很难想象这到底在干吗，因为 var_8 又还给 ecx 了，而且没有任何
;变化，是无用指令
.text:00011032        mov     ecx, [ebp+var_8]

;第一个参数也就是 PDRIVER_OBJECT 放入 edx，然后 edx 放
;入了 this+4 字节处。与这句对应的 C++语句是
;MyDriver::MyDriver(PDRIVER_OBJECT driver) : d_driver(driver)
;注意后面的参数初始化。也就是说，类对象的开头虚函数表之后
;接着就是第一个数据成员，但是静态成员不在其中
.text:00011035        mov     edx, [ebp+arg_0]
.text:00011038        mov     [ecx+4], edx

;清零 var_4，然后无条件跳到 loc_1104D。之后可以看到，var_4 其实
;是循环变量。下面的操作就是一个循环，填写分发函数。这里留给
;读者自己练习理解吧
.text:0001103B        mov     [ebp+var_4], 0
.text:00011042        jmp     short loc_1104D
.text:00011042
.text:00011044 loc_11044:
.text:00011044        mov     eax, [ebp+var_4]
.text:00011047        add     eax, 1
.text:0001104A        mov     [ebp+var_4], eax
.text:0001104A
.text:0001104D
.text:0001104D loc_1104D:
.text:0001104D        cmp     [ebp+var_4], 1Bh
.text:00011051        ja      short loc_11063
.text:00011051
.text:00011053        mov     ecx, [ebp+var_4]
.text:00011056        mov     edx, [ebp+arg_0]
.text:00011059        mov     dword ptr [edx+ecx*4+38h],
                              offset MyDriver__sDispatch
```

```
.text:00011061        jmp     short loc_11044
.text:00011061
.text:00011063
.text:00011063
.text:00011063 loc_11063:

;这里对应的是最后的 d_my_driver = this 一句。在这里也可以看到静
;态变量并不保存在对象内部，而是独立命名了一个全部变量
;名字就叫 MyDriver__d_my_driver
.text:00011063        mov     eax, [ebp+var_8]
.text:00011066        mov     MyDriver__d_my_driver, eax
.text:0001106B        mov     eax, [ebp+var_8]
.text:0001106E        mov     esp, ebp
.text:00011070        pop     ebp
.text:00011071        retn    4
.text:00011071
.text:00011071 MyDriver__MyDriver endp
```

在 C++编写的 Windows 内核程序的实际编译结果中，对象的开头部分就是虚函数表地址。

知识小结

总结：首先，这里编译的 C++的成员函数都使用_thiscall 调用方式，参数堆栈由函数自己恢复，ecx 传递 this 指针。其次，这里的类其实是一堆变量和函数的组合，静态数据成员以类名+双下画线+数据成员名命名，静态的和普通的成员函数都以类名+双下画线+成员函数名命名。第三，类的对象的开始地址处首先保存的是虚函数表（虚函数将在下一节中详细讨论），然后是普通的数据成员。

C++还有继承、虚拟、多态等种种特性，在下面的内容中会继续讨论。

9.3 了解更多的 C++特性

基础知识

虚函数在被调用的时候，只取决于那个对象的实际类型，而不取决于它的指针类型。这是一个非常有用的特性：只需要替换一个对象的实体（实际上一般都用工厂类来生成

实体，以至于在使用一个类及其派生类的时候根本不关心生成的对象究竟是什么类的对象），就可以完美地修改一部分功能，而其他的部分几乎完全不用担心这种细节的变化，只使用父类的指针操作，而不关心这些操作下层的实现。

代码分析

下面是上一节编写的 MyDriver 类的一个派生类。这个子类重写了分发函数，期待 IRP 下发的时候，调用的是新函数，打印一句“This is the dispatch function of MySubDriver!”。

```
class MySubDriver : public MyDriver
{
public:
        MySubDriver(PDRIVER_OBJECT driver) : MyDriver(driver){};
        virtual NTSTATUS OnDispatch(PDEVICE_OBJECT dev, PIRP irp)
        {
            DbgPrint("This is the dispatch function of MySubDriver!\ r\n");
            return STATUS_UNSUCCESSFUL;
        }
};
```

为了达到上面的目的，使用新类的对象。修改 DriverEntry：

```
extern "C" NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    MyDriver::d_my_driver = new MySubDriver(driver);
    return STATUS_UNSUCCESSFUL;
}
```

虽然 d_my_driver 的类型依然是 MyDriver，但是有 IRP 到来的时候，却确实会调用这里新写的 MySubDriver::OnDispatch 这个函数。当然，这是因为 OnDispatch 函数前面有 virtual 关键字；否则，被调用的还会是老版本的 MyDriver::OnDispatch。那么，这些是怎么实现的呢？

首先看 OnDispatch 这个虚函数是如何被调用的。这个函数在静态函数 sDispatch 中被调用：

```
.text:000110B0 MyDriver__sDispatch proc near
.text:000110B0
```

```
.text:000110B0 arg_0 = dword ptr  8
.text:000110B0 arg_4 = dword ptr  0Ch
.text:000110B0
.text:000110B0   push    ebp
.text:000110B1   mov     ebp, esp

;下面是为了调用 OnDispatch，而压入两个参数
.text:000110B3   mov     eax, [ebp+arg_4]
.text:000110B6   push    eax
.text:000110B7   mov     ecx, [ebp+arg_0]
.text:000110BA   push    ecx

;从这里开始寻找 OnDispatch 的地址。请注意，并不是直接调
;用 MyDriver__OnDispatch，而是从取得对象的内容开头的一个
;地址放入 eax 中，这个地址实际上是虚函数表
.text:000110BB   mov     edx, MyDriver__d_my_driver
.text:000110C1   mov     eax, [edx]
.text:000110C3   mov     ecx, MyDriver__d_my_driver

;这是实际调用 OnDispatch 函数的地方。这里取虚函数表中的第
;二个函数
.text:000110C9   call    dword ptr [eax+4]
.text:000110CC   pop     ebp
.text:000110CD   retn    8
.text:000110CD
.text:000110CD MyDriver__sDispatch endp
```

从上面内容可以看到，虚函数的调用和普通函数的调用之间的不同之处在于：普通函数是直接调用的；而虚函数是根据对象中保存的虚函数表地址（对象的开头就是虚函数表地址）找到虚函数表，然后寻找对应的函数。那么，派生类的虚函数表和基类的虚函数表一定有某种关系，才能在调用基类指针的函数时，能正确地指向对象的真实虚函数上去。下面是派生类的构造函数，从这里可以看出虚函数表地址是如何初始化的。

```
.text:00011110 MySubDriver__MySubDriver proc near
.text:00011110
.text:00011110 var_4 = dword ptr -4
.text:00011110 arg_0 = dword ptr  8
.text:00011110

;普通函数入口流程
.text:00011110       push    ebp
.text:00011111       mov     ebp, esp
```

```
;ecx 是对象的 this 指针。这里入栈后，放入 var_4 中备份了一下
;但是不明白为何这里 ecx 要入栈，而且也没有看到它出栈。但是
;这并不影响堆栈的平衡，因为最后恢复了 esp。可能是调试版本
;的没有被优化掉的多余的代码
.text:00011113        push    ecx
.text:00011114        mov     [ebp+var_4], ecx

;push 参数，然后调用了 MyDriver__MyDriver，var_4 再放入 ecx
;中，等于 ecx 没变，还是 this 指针。等于这里初始化了基类
.text:00011117        mov     eax, [ebp+arg_0]
.text:0001111A        push    eax
.text:0001111B        mov     ecx, [ebp+var_4]
.text:0001111E        call    MyDriver__MyDriver
.text:0001111E

;然后初始化派生类。这里写入了派生类的虚函数表，实际上替换了
;基类的虚函数表
.text:00011123        mov     ecx, [ebp+var_4]
.text:00011126        mov     dword ptr [ecx],
                              offset const MySubDriver::`vftable'
.text:0001112C        mov     eax, [ebp+var_4]

;接着恢复 esp，ebp，在返回的时候平衡堆栈
.text:0001112F  mov     esp, ebp
.text:00011131          pop     ebp
.text:00011132          retn    4
.text:00011132
.text:00011132 MySubDriver__MySubDriver endp
```

一个类派生出另一个类之后，虚函数表会被替换成一份新的表。为了观察这两个表，我在 MyDriver 类中增加了一个新的虚函数。

```
virtual void DoSomething(){};
```

这个函数没有被重载，这样，用 IDA 能看到虚函数表的结构如下：

```
; const MyDriver::`vftable'
.rdata:0001202C dd offset MyDriver__DoSomething
.rdata:00012030  dd offset MyDriver__OnDispatch

;const MySubDriver::`vftable'
.rdata:00012034  dd offset MyDriver__DoSomething
.rdata:00012038  dd offset MySubDriver__OnDispatch
```

在 C++编写的 Windows 内核程序的反汇编结果中，虚函数表示父类和派生类各有一份，没有共用的部分。

知识小结

结论：基类的虚函数表和派生类的虚函数表各有一份。在派生类的虚函数表中，没有重载的基类虚函数的地址和基类的虚函数表相同，而重载过的虚函数则是自己独特的地址。

但是要注意的是，这些并非是一成不变的规则，不同的 C++编译器可能有不同的实现方法。

这些虚函数表是 C++编译器生成的，实际上这反而给 HOOK 这些函数带来了方便。在一些安全软件或者病毒中，使用了修改虚函数表来 HOOK 某些调用的方法。

C++的实现原理可以写成一本砖头厚的大书。但是本书没有足够的篇幅那样做，获取解决问题所需要的知识才是本书的目的。现在读者已经掌握了研究的方法，剩下的问题可以自己开始研究了。

第 10 章　继续探索 Windows 内核

10.1 探索 Windows 已有内核调用

基础知识

下面用本书前面介绍的知识来做一些有用的事情。Windows 从 2000 发展到 XP 后，XP DDK 中出现了一些新的调用。内核程序开发者有时会发现，这些调用非常有用（这也是这些新调用产生的原因），但是如果使用这些调用，会导致驱动在 Windows 2000 下无法使用。目前 Windows 2000 的用户依然很多，存在这样的可移植性问题是非常遗憾的，退一步的方案是 Windows 2000 下限制某些功能。在程序中动态加载系统调用，并小心地判断当前的版本。在 Windows 2000 的情况下，一些功能被跳过。这样比前者好，但是依然不是最理想的解决方案。

有一些人开始使用未公开（Undocumented）的解决方案。未公开的解决方案的危害就是，可能不兼容未来的操作系统。但是由于现在这样做仅仅是针对一个过去版本的操作系统，问题就不复存在了。在新版本的操作系统上，调用新的系统调用，而在某个已经存在而且不会再变的操作系统版本上，调用未公开的方法。只要低版本操作系统测试无问题，以后的也不会有问题。

这里的所谓未公开的解决方案就是，在新版本的操作系统上，把新出现的功能调用进行调试，了解其实现的方法。然后在我们自己开发的内核程序中，实现同样的或者等价的功能，同时检测当前操作系统为低版本时，调用这些特殊的代码。

下面举两个例子。

```
PDEVICE_OBJECT
    IoGetDeviceAttachmentBaseRef(
    IN PDEVICE_OBJECT  DeviceObject);
```

这个调用获得设备栈底的设备，这对过滤驱动非常有用。

为了自己实现它，首先必须要看到这些调用的汇编代码。这并不难，读者可以用 WinDbg 调试运行 Windows，WinDbg 可以自动下载对应的符号表。操作简单，随时可以设置断点调试。

调试 Windows 需要两台机器，或者在一台计算机上使用虚拟机，用实际的串口线或者管道模拟的串口连接，并经过一系列的设置。具体请参考前面的“8.2 学习用

WinDbg 进行调试”一节。

上机实践

我很希望调试 Windows XP 下这两个函数。不过我的手头只有一台被调试的 Vista，所以下面实际上是 Vista 的调试结果。现在打开 WinDbg，在命令输入框中输入：

```
u IoGetDeviceAttachmentBaseRef
```

显示如下：

```
kd> u IoGetDeviceAttachmentBaseRef
nt!IoGetDeviceAttachmentBaseRef:
818c151c 8bff    mov     edi,edi
818c151e 55      push    ebp
818c151f 8bec    mov     ebp,esp
818c1521 53      push    ebx
818c1522 56      push    esi
818c1523 6a0a    push    0Ah
818c1525 59      pop     ecx
818c1526 ff1500118081    call
dword ptr [nt!_imp_KeAcquireQueuedSpinLock (81801100)]
```

现在看到的是 IoGetDeviceAttachmentBaseRef 的开头部分。我们已经看到了地址为 818c151c，那么要看完整的部分也简单了，单击主菜单“View”→“Disassembly”，出现 Disassembly 窗口，在上部的 offset 输入框中输入 818c151c，IoGetDeviceAttachmentBaseRef 的代码如下：

```
nt!IoGetDeviceAttachmentBaseRef:
818c151c 8bff    mov     edi,edi      ;无意义指令
818c151e 55      push    ebp          ;保存 ebp
818c151f 8bec    mov     ebp,esp      ;保存 esp
818c1521 53      push    ebx          ;保存 ebx
818c1522 56      push    esi          ;保存 esi
818c1523 6a0a    push    0Ah

;把 0ah 当做第一个参数传给 KeAcquireQueuedSpinLock
818c1525 59      pop     ecx
818c1526 ff1500118081  call
dword ptr [nt!_imp_KeAcquireQueuedSpinLock (81801100)]
```

```
;bl = KeAcquireQueuedSpinLock 的返回值
818c152c 8ad8    mov         bl,al

;取得唯一的一个参数 DEVICE_OBJECT
818c152e 8b4508  mov      eax,dword ptr [ebp+8]
818c1531 e8d3ffffff       call
nt!IopGetDeviceAttachmentBase+0x4 (818c1509)
818c1536 8bf0    mov         esi,eax
818c1538 8bce    mov         ecx,esi
818c153a e8dd5ff8ff  callnt!ObfReferenceObject (8184751c)
818c153f 6a0a    push        0Ah
818c1541 8ad3    mov         dl,bl
818c1543 59      pop         ecx
818c1544ff15fc108081 call
dword ptr [nt!_imp_KeReleaseQueuedSpinLock (818010fc)]
818c154a 8bc6    mov      eax,esi
818c154c 5e      pop      esi
818c154d 5b      pop      ebx
818c154e 5d      pop      ebp
818c154f c20400  ret 4
```

同时，IopGetDeviceAttachmentBase 在调试器中的汇编代码如下：

```
nt!IopGetDeviceAttachmentBase:
818c1500 eb07    jmp
nt!IopGetDeviceAttachmentBase+0x4 (818c1509)
818c1502 8bc1    mov eax,ecx
818c1504 90      nop
818c1505 90      nop
818c1506 90      nop
818c1507 90      nop
818c1508 90      nop
818c1509 8b88b0000000
                 mov      ecx,dword ptr [eax+0B0h]
818c150f 8b4918  mov      ecx,dword ptr [ecx+18h]
818c1512 85c9    test     ecx,ecx
818c1514 75ec    jne
nt!IopGetDeviceAttachmentBase+0x2 (818c1502)
818c1516 c3      ret
```

要理解上面的代码，现在可能还有些问题。这在下节中描述。

10.2　自己实现 XP 的新调用

读者阅读 Vista 下的 IopGetDeviceAttachmentBase 可能有点小麻烦，因为 Iop 系列的函数是内部函数，而微软的内部函数系统常常用 fast call 的方式。这种方式与前面的 std call 方式几乎完全相同，唯一的区别是，前两个参数不被放到堆栈中传入，而是放入 ecx 和 edx 中，ecx 中保存第一个参数，edx 中保存第二个参数。

下面尝试自己实现上面函数的等价函数，来用于 Windows 2000 操作系统上。

fast call 函数的参数传递方法是 ecx 传递第一个参数，edx 传递第二个参数，其他的使用堆栈传递。

10.2.1　对照调试结果和数据结构

这就是 IopGetDeviceAttachmentBase 这个函数返回的时候不需要恢复堆栈的原因，因为参数没有通过堆栈传递。此外令人疑惑的是开头的 jmp，这样，通过 ecx 传递参数就被跳过了。我只能理解为，这个函数应该是从 818c1502 开始的。

IopGetDeviceAttachmentBase 的被调方式是：

```
; 取得唯一的一个参数 DEVICE_OBJECT
818c152e 8b4508      mov     eax,dword ptr [ebp+8]
818c1531 e8d3ffffff  call    nt!IopGetDeviceAttachmentBase+0x4
(818c1509)
```

这并非直接调用 nt!IopGetDeviceAttachmentBase 这个函数，而是直接把参数 mov 到 eax 中，然后 jmp 到了 818c1509 处。这些代码在 Vista 下经常出现。不像是一般的编译结果，像是经过某种特殊的优化后的结果。

818c1509 处开始的代码就很简单了：

```
818c1509 8b88b0000000    mov     ecx,dword ptr [eax+0B0h]
818c150f 8b4918          mov     ecx,dword ptr [ecx+18h]
818c1512 85c9            test    ecx,ecx
818c1514 75ec            jne     nt!IopGetDeviceAttachmentBase+0x2
(818c1502)
818c1516 c3              ret
```

eax 是传入参数，也就是 PDEVICE_OBJECT 的指针。mov ecx,dword ptr [eax+0B0h] 中，DEVICE_OBJECT 的 B0 是一个让人有些眼熟的位置。下面是 XP 下 DEVICE_OBJECT 的结构，内容如下：

```
    struct _XP2600_2180_DEVICE_OBJECT /* sizeof 000000B8 184 */
    {
    /* off 0x00000000 */ short    Type;
    /* off 0x00000002 */ unsigned short   Size;
    /* off 0x00000004 */ long     ReferenceCount;
    /* off 0x00000008 */ struct _XP2600_2180_DRIVER_OBJECT*    Driver
Object;
    /* off 0x0000000C */ struct _XP2600_2180_DEVICE_OBJECT*    NextDevice;
    /* off 0x00000010 */ struct _XP2600_2180_DEVICE_OBJECT*    Attached
Device;
    /* off 0x00000014 */ struct _XP2600_2180_IRP* CurrentIrp;
    /* off 0x00000018 */ struct _XP2600_2180_IO_TIMER*     Timer;
    /* off 0x0000001C */ unsigned long    Flags;
    /* off 0x00000020 */ unsigned long    Characteristics;
    /* off 0x00000024 */ struct _XP2600_2180_VPB* Vpb;
    /* off 0x00000028 */ void*    DeviceExtension;
    /* off 0x0000002C */ unsigned long    DeviceType;
    /* off 0x00000030 */ char     StackSize;
    /* off 0x00000034 */ union _XP2600_2180__unnamed_000001D5 Queue;
    /* off 0x0000005C */ unsigned long    AlignmentRequirement;
    /* off 0x00000060 */ struct _XP2600_2180_KDEVICE_QUEUE     DeviceQueue;
    /* off 0x00000074 */ struct _XP2600_2180_KDPC Dpc;
    /* off 0x00000094 */ unsigned long    ActiveThreadCount;
    /* off 0x00000098 */ void*    SecurityDescriptor;
    /* off 0x0000009C */ struct _XP2600_2180_KEVENT   DeviceLock;
    /* off 0x000000AC */ unsigned short   SectorSize;
    /* off 0x000000AE */ unsigned short   Spare1;
    /* off 0x000000B0 */ struct _XP2600_2180_DEVOBJ_EXTENSION*DeviceObject
Extension;
    /* off 0x000000B4 */ void*    Reserved;
    };
```

这个头文件是笔者在驱动开发网上下载的，里面有部分数据成员是微软提供的 WDK 中未公开的。如果读者不知道这个结构，请直接用指针移动 0xb0 字节来获取就可以了，没有必要关心这个结构。请注意由于许多数据是未公开的，所以不同版本的 Windows 可能不同。

下一步是 mov ecx,dword ptr [ecx+18h]，可见[eax+0B0h]本身也是结构体指针。再查一下这个结构：

```
typedef struct _XP2600_2180_DEVOBJ_EXTENSION /* sizeof 0000002C 44 */
{
/* off 0x00000000 */ short    Type;
/* off 0x00000002 */ unsigned short   Size;
/* off 0x00000004 */ struct                    _XP2600_2180_DEVICE_OBJECT*
DeviceObject;
/* off 0x00000008 */ unsigned long    PowerFlags;
/* off 0x0000000C */ struct
_XP2600_2180_DEVICE_OBJECT_POWER_EXTENSION*   Dope;
/* off 0x00000010 */ unsigned long    ExtensionFlags;
/* off 0x00000014 */ void*    DeviceNode;
/* off 0x00000018 */ struct _XP2600_2180_DEVICE_OBJECT*    AttachedTo;
/* off 0x0000001C */ long     StartIoCount;
/* off 0x00000020 */ long     StartIoKey;
/* off 0x00000024 */ unsigned long    StartIoFlags;
/* off 0x00000028 */ struct _XP2600_2180_VPB* Vpb;
}XP2600_2180_DEVOBJ_EXTENSION,*PXP2600_2180_DEVOBJ_EXTENSION;
```

0x18 处是 AttachedTo，而且实践表明，从 XP 到 Vista 微软都没有改变过这个结构。那么 Windows 2000 下是否是一样的呢？这个可以自己验证一下：在 Windows 2000 的电脑上运行 10.2.2 节的代码，看看是否正常。

10.2.2　写出 C 语言的对应代码

自己用 C 语言实现同样的功能。没有头文件的时候，读者得自己定义结构中 0x80 和 0x18 这样的位置，当然定义什么那是无所谓的，只要目的是明确的。在有以上头文件的时候，可以这样实现：

```
PDEVICE_OBJECT IopGetDeviceAttachmentBase(
    PDEVICE_OBJECT device)
{
    PXP2600_2180_DEVICE_OBJECT mydevice =
        (PXP2600_2180_DEVICE_OBJECT)device;
    PXP2600_2180_DEVOBJ_EXTENSION my_extension =
        (PXP2600_2180_DEVOBJ_EXTENSION)mydevice->
    DeviceExtension;
    if(my_extension->AttachedTo == NULL)
        return device;
    else
        return  IopGetDeviceAttachmentBase(
        (PDEVICE_OBJECT)my_extension->AttachedTo);
}
```

然后是 IoGetDeviceAttachmentBaseRef，从前文的汇编看，流程是这样的：

第一步，call dword ptr [nt!_imp_KeAcquireQueuedSpinLock (81801100)]，参数为 0xA。

第二步，调用 IopGetDeviceAttachmentBase。

第三步，call nt!ObfReferenceObject (8184751c)，目的是对这个设备记一次引用。

第四步，call dword ptr [nt!_imp_KeReleaseQueuedSpinLock (818010fc)]，这是对前面的 KeAcquireQueuedSpinLock 调用的一个释放。

在以上的调用中，ObfReferenceObject 在 Windows 2000 下存在，IopGetDeviceAttachmentBase 我们自己实现了；KeAcquireQueuedSpinLock 和 KeReleaseQueuedSpinLock 是为了同步设备链，考虑多线程的情况，我在遍历设备链表的时候，显然不希望其他线程操作这个设备链表。KeAcquireQueuedSpinLock 这个调用在 Windows 2000 下也没有，但是仅仅是阻止线程切换的话，我们可以通过提高中断级来实现。这按理和 KeAcquireQueuedSpinLock 效果是一样的。

下面是我们自己实现的功能类似的函数。请在 Windows 2000 上测试它。

```
PDEVICE_OBJECT
    IoGetDeviceAttachmentBaseRef(
    IN PDEVICE_OBJECT  DeviceObject)
{
    KIRQL irql;
    PDEVICE_OBJECT baseDevice;
    irql =  KeRaiseIrqlToDpcLevel();
    baseDevice = IopGetDeviceAttachmentBase(DeviceObject);
    ObReferenceObject(baseDevice);
    KeLowerIrql(irql);
    return baseDevice;
}
```

10.3 没有符号表的情况

基础知识

没有符号表的情况是常见的。如果你对一个病毒、木马的内核组件产生了兴趣，很想了解它的工作原理，但是显然病毒作者没有义务为你提供符号表。作为必要的研究，你依然可以尝试去了解这些代码（**但是如果要用于商业的软件，则请首先获取知识产权所有者的授权**）。

幸运的是，即使没有符号表，IDA 依然可以工作，只是由于没有符号表，你可能不知道 DriverEntry 函数在哪里。但是只要进入了这个函数，剩下的研究就比较简单了。你可以从 DriverEntry 看到 Dispatch Routines，然后依次地看下去。

虽然说看不见一个叫做 DriverEntry 的入口，但是你总是可以在 IDA 反汇编的结果中，看到 INIT 段中的 start 过程。具体的一个例子如下：

```
INIT:00014000 INIT  segment para public 'CODE' use32
INIT:00014000    assume cs:INIT
INIT:00014000    ;org 14000h
INIT:00014000    assume es:nothing, ss:nothing,
                 ds:_data, fs:nothing, gs:nothing
INIT:00014000
INIT:00014000 ; ** ** S U B R O U T I N E ****
INIT:00014000    public start
INIT:00014000 start  proc near
INIT:00014000    mov     eax, dword_13000
INIT:00014005    test    eax, eax
INIT:00014007    mov     ecx, 0BB40E64Eh
INIT:0001400C    jz      short loc_14012
INIT:0001400C
INIT:0001400E    cmp     eax, ecx
INIT:00014010    jnz     short loc_1402B
INIT:00014010
INIT:00014012
INIT:00014012    loc_14012:
INIT:00014012    mov     eax, ds:KeTickCount
INIT:00014017    mov     eax, [eax]
INIT:00014019    xor     eax, offset dword_13000
INIT:0001401E    mov     dword_13000, eax
INIT:00014023    jnz     short loc_1402B
INIT:00014023
INIT:00014025    mov     dword_13000, ecx
INIT:00014025
INIT:0001402B
INIT:0001402B    loc_1402B:
INIT:0001402B                                        ; INIT:0001402B
jmp    sub_11028 ; 注意这条 jmp 指令
INIT:0001402B
INIT:0001402B start  endp
```

前面的处理，请用自己在前面学到的知识进行分析。这个 start 过程在最后出现一个 jmp 指令，如上面重点注释的 jmp sub_11028。一般地说，这条 jmp 指令跳到 DriverEntry 函数处开始执行。换句话说，sub_11028 就是 DriverEntry 函数。

不过这是 check 版的情况。free 版的情况反而要简单得多，请自己动手分析。

此时，请在 sub_11028 上单击鼠标右键，在弹出的菜单中选择“Rename”，然后将 sub_11028 改名为 DriverEntry。这样，即使是在完全没有符号表的情况下，你最终也能逐步标出各个公用的函数接口。但是剩余的内部函数，就需要你去逐步猜测了。无论如何，这是一个在 IDA 上不断标记，最终还原出大部分关键代码的过程。

一般地说，如果驱动的作者没有对代码进行特殊的处理（请参照后面的第 16、17 章），这种方法持续地进行，反工程一个驱动程序的难度并不大。

思考与练习

下面就是找到的 DriverEntry 函数。这些代码非常简单，作为练习，请先不要看后面的正确答案，写出对应的 C 语言代码。

```
.text:00011028
.text:00011028 ; ***** S U B R O U T I N E ****
.text:00011028
.text:00011028 ; Attributes: bp-based frame
.text:00011028
.text:00011028 ; int __stdcall sub_11028(PDRIVER_OBJECT DriverObject,
int)
.text:00011028 sub_11028 proc near        .text:00011028
.text:00011028 DeviceName    = UNICODE_STRING ptr -0Ch
.text:00011028 DeviceObject  = dword ptr -4
.text:00011028 DriverObject  = dword ptr  8
.text:00011028
.text:00011028       push    ebp
.text:00011029       mov     ebp, esp
.text:0001102B       sub     esp, 0Ch
.text:0001102E       push    offset    SourceString    ;    "\\Device\
MyFilterCDO"
.text:00011033       lea     eax, [ebp+DeviceName]
.text:00011036       push    eax             ; DestinationString
.text:00011037       call    ds:RtlInitUnicodeString
.text:0001103D  lea          eax, [ebp+DeviceObject]
.text:00011040       push    eax             ; DeviceObject
.text:00011041       push    0               ; Exclusive
.text:00011043       push    100h            ; DeviceCharacteristics
```

```
.text:00011048   push    22h             ; DeviceType
.text:0001104A   lea     eax, [ebp+DeviceName]
.text:0001104D   push    eax             ; DeviceName
.text:0001104E   push    0               ; DeviceExtensionSize
.text:00011050   push    [ebp+DriverObject] ; DriverObject
.text:00011053   call    ds:IoCreateDevice
.text:00011059   leave
.text:0001105A   retn    8
.text:0001105A
.text:0001105A   sub_11028    endp
```

实际上，以上驱动是我随手写的一个简单驱动程序，只是生成一个设备。C 代码的正确答案如下：

```
NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    UNICODE_STRING name_str;
    PDEVICE_OBJECT my_cdo;
    NTSTATUS status;
    RtlInitUnicodeString( &name_str, L"\\Device\\MyFilterCDO" );
    status = IoCreateDevice(
                                driver,
                                0,
                                &name_str,
                                FILE_DEVICE_UNKNOWN,
                                FILE_DEVICE_SECURE_OPEN,
                                FALSE,
                                &my_cdo );
    return status;
}
```

10.4　64 位操作系统下的情况

基础知识

现在在 PC 上使用 64 位操作系统的情况还不多见（服务器会用得比较多），这是因为有太多的软件在 64 位操作系统上不兼容。虽然 AMD64 架构使我们即使在 64 位操作

系统的情况下依然可以使用 32 位的应用程序，但是内核代码却必须全部重新编译。

常见的 64 位操作系统有 64 位版本的 Windows XP、Windows 2003 Server 和 Vista，我手头只有一份 64 位的 Windows 2003 Server，因此也只能用它来看看。没有看到 64 位版本的 Vista，是一种遗憾，请有兴趣的读者自己研究。

关于位数（8 位、16 位、32 位、64 位和更多的位）

CPU 的位数似乎取决一次运算能运算的数据长度，也表现为大部分寄存器的位数，以及地址线和数据线的条数等。CPU 如果有 32 条数据线，则一次就可以传输 32 位的数据。同时一般此时大部分寄存器也是 32 位的，一次操作，比如一个 add 指令，可以一次操作 32 位的数据，此时这个 CPU 应该是 32 位的 CPU。x86 的 CPU 有一个发展的过程，一般高级的 CPU 都兼容低级 CPU 的指令，所以 32 位的 CPU 也同样接受 16 位操作。

当今 CPU 发展的最高等级是 64 位 CPU，实际上完全不是如此。电视游戏机 PS2 就是使用了高达 128 位的 CPU。只是不是 x86 架构的 CPU。x86 是 PC 上最常见的最占统治地位的 CPU 架构，但是绝对不是唯一的架构。

实际上更精确的说法，应该是 IA32 架构，这个架构包含对此类 CPU 支持的所有指令的精确描述。

IA64 是一种和 x86 不兼容的 64 位架构，早在 AMD64 架构出现之前就已经出现了，但是不能用于 PC，所以很少有普通使用者有兴趣去了解它。但是 Windows 内核程序可以编译成 IA64 版本。

AMD64 位是第一种能兼容 IA32 的 64 位架构。此后 Intel 引入了相对应的 EM64T 架构，与 AMD64 差别很少。

由于兼容性，所以 AMD64 平台上显然可以运行 32 位版本的 Windows XP。实际上现在许多 PC 使用 64 位的 CPU，却运行着 32 位的 Windows。我现在写作时用的笔记本电脑正是如此。

64 位版本的 Windows XP 和 32 位版本不同，但是依然可以支持 32 位上的大部分应用程序。有兴趣的读者可以安装 64 位版本的操作系统，但是可能碰到许多应用了内核组件的软件不兼容。

64 位版本的 Windows 和 32 位版本的内核程序不兼容。

下面以 IofCallDriver 这个函数为例。这个函数非常重要，在后面的示例中会常常涉及。如果需要自己动手调试的话，则需要在虚拟机上安装一个 64 位版本的操作系统。其他 VMware 和 WinDbg 的设置参照前面 32 位操作系统的情况，没有任何不同。

连接上调试器，输入 u IofCallDriver，打开 Disassembly 窗口。输入 u IofCallDriver 指令显示的开始地址，就可以看见 IofCallDriver 的汇编代码。详细分析见下一节。

10.4.1　分析 64 位操作系统的调用

代码分析

下面的代码也是在 WinDbg 中调试 IofCallDriver 的结果。只是因为 64 位情况下地址都非常长，放在这里打印出来会占据太多的地方，所以去掉了所有的地址和机器码。请注意每一句汇编之前的注释。

```
nt!IofCallDriver:

;腾出堆栈中内部变量空间，也腾出调用后面可能的其他函数的参数
;栈空间。我曾经很迷惑这种写法，具体看后面的解说
sub     rsp,28h

;把 nt!IofCallDriver 的值放入 rax 中
mov     rax,qword ptr [nt!pIofCallDriver (fffff800`01179600)]
;比较 rax，其实也就是 nt!pIofCallDriver 是否为 0
test    rax,rax
;不是 0 的情况，跳转到 fffff800`01093bbb，那段代码写在下面
jne     nt!IoCallDriver+0x10 (fffff800`01093bbb)

;如果 pIofCallDriver 为 0，走这里。把 rdx+43h，也就是第二个参
;数所指的结构的 43h 位置处的数字减 1。我想应
;该是 Irp->CurrentLocation--;
dec     byte ptr [rdx+43h]

;如果减 1 后小于等于 0，实际上已经出错。这里会跳转到
;一个地方去调用 KiBugCheck3 直接蓝屏
cmp     byte ptr [rdx+43h],0
jle     nt!IoCallDriver+0x25 (fffff800`01093bc7)

;找到 DriverObject 对应的分发函数的
;开始地址放入 eax 中，应该是 device->DriverObject->
;DispatchFunctions[irpsp->MajorFunction]
mov     rax,qword ptr [rdx+0B8h]
sub     rax,48h
```

```
    mov     qword ptr [rdx+0B8h],rax
    mov     qword ptr [rax+28h],rcx
    mov     r8,qword ptr [rcx+8]
    movzx   eax,byte ptr [rax]

    ;自己恢复堆栈空间
    add     rsp,28h

    ;可以假设这是在调用一个分发函数。奇怪的是为何没有用 call
    ;因为这个函数试图直接返回到调用这个函数之前的地址
    ;那样用 call 就浪费了
    jmp     qword ptr [r8+rax*8+70h]

    ;无意义的空指令，它们把不同函数分隔开
    nop
    nop
    nop
    nop
    nop
    nop
    nop

    ... ... ;这里跳过了很多其他函数的指令

fffff800`01093bbb:
    ;这里的本质就是调用 pIofCallDriver（事先已经保存在 rax 中）
    ;此外，还有一段调用 KiBugCheck3 来直接蓝屏的代码
    mov     r8,qword ptr [rsp+28h]
    add     rsp,28h
    jmp     rax
    xor     r9d,r9d
    xor     r8d,r8d
    lea     ecx,[r9+35h]
    call    nt!KiBugCheck3 (fffff800`01041660)
    int     3
```

知识小结

如果这段代码看得不是很懂，读者可以在之后比较一下在 32 位版本 Vista 上的调试结果（本书第 13 章将详细介绍 Hook IofCallDriver 函数的技术）。上面看到一些关于 64 位汇编有趣的地方，在 64 位下，rsp 这个寄存器取代了 esp。rsp 与 esp 的关系和 esp 与 sp 的关系类似，同理还有 rax、rcx、rdx 等。此外出现了新的普通寄存器 r8、r9。实际

上，标准的 32 位 x86 架构包括 8 个通用寄存器，AMD 在 x86-64 中又增加了 8 组（r8~r15），将寄存器的数目提高到了 16 组。x86-64 寄存器默认为 64 位。

要了解以下的同一个寄存器的不同部分：rax（64 位）、eax（低 32）、ax（低 16 位）、al（低 8 位）、ah（8 到 15 位）。同样，对 r8~r15，相应的有 r8、r8d、r8w 和 r8b。

此外，内核参数的传递，从这个函数只能看出是用 rcx 和 rdx 传递了前两个参数，但是后面的参数是如何传递的还未知。这是有原因的，因为 IofCallDriver 本来就只有两个参数。接下来本书通过实践来解决这个问题。

10.4.2　深入了解 64 位内核调用参数传递

上机实践

为了上节末尾的问题，读者可以自己编译一个 64 位的驱动程序来试试。我曾试图查阅各种资料，但是没有找到 64 位版本的 Windows 内核函数传递参数的方法说明。也许不同版本的操作系统上也会有不同，不是很有把握可以找到确切的规则，还是眼见为实比较好。

所以，我决定自己编译一个 64 位版本的驱动程序进行 IDA 反汇编来了解这个情况。既然要了解参数的传递过程，那么显然函数需要各种参数，而且它们必须都被使用了。

```
ULONG MyTestFunc(
    PDEIVCE_OBJECT device1,
    ULONG ulong1,
    CHAR char1,
    CHAR char2,
    PCHAR str1,
    SHORT short1,
    SHORT short2,
    PVOID void1,
    PVOID void2,
    PVOID void3)
{
    PDEVICE_OBJECT mydevice1 = device1;
    ULONG myulong1 = ulong1;
    CHAR mychar1 = char1;
    CHAR mychar2 = char2;
    PCHAR mystr1 = str1;
    SHORT myshort1 = short1;
```

```
    SHORT myshort2 = short2;
    PVOID void1 = void1;
    PVOID void2 = void2;
    PVOID void2 = void3;
    return 0;
}
```

上面这个函数的参数无论是数量还是种类都非常的丰富，而且有同样多的内部变量，足够我们看出参数的传递过程。为了避免这些无用的参数编译的时候被优化掉，所以我将编译 AMD64 的 check 版本的驱动程序来避免优化。我曾认为 ULONG 和 int 会是 64 位数据，但是却发现只有指针是 64 位数据，ULONG 依然是 32 位。char、short 和 int 的长度也和 32 位下的情况没有区别。此外有一个特殊的 ULONG_PTR 类型是 64 位。

反汇编的结果有点出乎我的意料，本来以为传递参数的时候，除了 rcx、rdx、r8、r9 之外，可能还会用到 r10、r11…，但是实际情况并不是这样的，前 4 个参数用 rcx、rdx、r8、r9 传递，后面的通过栈传递。

```
sub_11000       proc near
; 函数调用时的堆栈状况。0 为栈顶。从-48 开始如下
var_48  = byte ptr -48h         ; 这里是 IDA 的栈标识。这是栈顶
var_44  = word ptr -44h
var_40  = qword ptr -40h
var_38  = qword ptr -38h
var_30  = word ptr -30h
var_2C  = byte ptr -2Ch
var_28  = qword ptr -28h
var_20  = qword ptr -20h
var_18  = dword ptr -18h
var_10  = qword ptr -10h
arg_0   = qword ptr  8    ;离栈顶第 8 字节是第一个参数（device1）
                          ;的空间
arg_8   = dword ptr  10h  ;第 16 字节（10h=16）为第二个参数（ulong1）的空
                          ;间，依此类推。可见虽然前 4 个参数通
arg_10  = byte ptr  18h   ;过寄存器
                          ;传递，但是在栈中还是有空间的。只是不
                          ;通过栈传值
arg_18  = byte ptr  20h
arg_20  = qword ptr  28h
arg_28  = word ptr  30h
arg_30  = word ptr  38h
```

```
arg_38  = qword ptr  40h
arg_40  = qword ptr  48h
arg_48  = qword ptr  50h
    mov     [rsp+arg_18], r9b
    mov     [rsp+arg_10], r8b
    mov     [rsp+arg_8], edx
    mov     [rsp+arg_0], rcx
    sub     rsp, 48h
    mov     rax, [rsp+48h+arg_0]
    mov     [rsp+48h+var_40], rax
    mov     eax, [rsp+48h+arg_8]
    mov     [rsp+48h+var_18], eax
    movzx   eax, [rsp+48h+arg_10]
    mov     [rsp+48h+var_2C], al
    movzx   eax, [rsp+48h+arg_18]
    mov     [rsp+48h+var_48], al
    mov     rax, [rsp+48h+arg_20]
    mov     [rsp+48h+var_28], rax
    movzx   eax, [rsp+48h+arg_28]
    mov     [rsp+48h+var_44], ax
    movzx   eax, [rsp+48h+arg_30]
    mov     [rsp+48h+var_30], ax
    mov     rax, [rsp+48h+arg_38]
    mov     [rsp+48h+var_38], rax
    mov     rax, [rsp+48h+arg_40]
    mov     [rsp+48h+var_20], rax
    mov     rax, [rsp+48h+arg_48]
    mov     [rsp+48h+var_10], rax
    xor     eax, eax
    ;这里只恢复了内部栈空间，所以函数调用者要负责恢复参数
    ;调用空间
    add     rsp, 48h
    retn
sub_11000       endp
```

知识小结

可以获得以下的信息：前 4 个参数分别由 rcx、rdx、r8、r9 来传递，但是调用者依然要根据这 4 个参数的大小来分配它们应有的堆栈空间（只是不用写入值）。而且这 4 个参数无论大小，都必须占用一个 64 位的寄存器。在堆栈空间中，也是一样，无论每个参数多大，都要实际占据 8 个字节（64 位）的栈空间大小。

64 位版本的 Windows 系统的内核函数由 rcx、rdx、r8、r9 分别传递第 1~4 个参数，如果有更多的参数则通过堆栈传递。

此外函数内部自己并不复原参数空间，参数空间由外面调用者恢复，但是见不到调用者这样做：

```
push a1
push a2
call myfunction1
pop a2
pop a1
push b1
push b2
call myfunction2
pop b2
pop b1
```

原因是这样做实在太烦琐了，每调用一个函数都要 push pop。所以，每个函数一进入，首先就分配足够多的堆栈空间，供自己做内部变量，以及需要传入内部调用其他函数时使用。

```
sub     rsp, xx
```

这时候已经有足够大的栈空间了，调用任何函数的时候，只要把必要的参数直接 mov 进栈空间中，然后调用函数即可。并不用马上恢复，调用其他函数的时候，这些栈空间还可以重复利用，只有到了本函数最终返回之前，我们要恢复堆栈。

```
add     rsp, xx
```

这样做比反复地 push、pop 效率高多了，大概这也是这种调用方法的优秀之处吧。读者会发现几乎每个内核函数，都由 sub 作为开始，add 作为结束。

深入篇

修改内核

这是本书的第四部分。读者已经尝试过探索 Windows 内核程序，并尝试阅读反汇编代码。那么接下来，必须掌握修改内核的方法。每一个 Windows 内核程序，都可以看做 Windows 内核本身的一个“补丁”。有时只需要独立存在，就能起到它的作用；有时却必须对已有的内核二进制代码进行部分修改。本部分包括第 11~13 章，主要介绍的是内核 Hook。

第 11 章　机器码与反汇编引擎

11.1　了解 Intel 的机器码

这里有几个常见的容易被错误认识的问题：汇编助记符与机器指令是一一对应的关系吗？一条汇编指令就对应着一个机器码吗？答案是否定的。比如 jmp 之类的指令，实际上可能有多种机器码实现形式。唯一肯定的是，所有的汇编指令最终都变成由 0 和 1 组成的机器码。我们看到的汇编指令，实际上是 WinDbg 和 IDA 一类的反汇编软件对机器指令的再次识别，反向翻译成汇编指令（所谓反汇编）的结果。这个结果并不一定保证是完全正确的。

本节专门讲述如何从机器码中解析指令，换句话说，这是介绍机器指令格式的部分。最精确的描述需要从 Intel 和 AMD 的指令手册中获得，但是本节讲述的知识，有助于读者快速获得基础知识，是入门的捷径。

11.1.1　可执行指令与数据

你能区分一段“数据”和一段“可执行指令”吗？除了实际的程序执行过程之外，并没有任何方法能保证一段 0 和 1 组成的数据是可执行的指令，或者实际上并不是用来执行的指令，而是一段有其他用途的数据。甚至错误的程序，也完全可能在执行过程中，跳到根本不能执行的地方继续执行——通常都会引发系统崩溃。

x86 所有指令的机器码长度不定，且连续排列，因此读取机器码的唯一方法是从头开始逐条解析指令。

x86 和 x64 的机器码的一个特点就是，每条指令的长度是不固定的，指令和指令之间并没有明确的分隔，你可以从第一字节开始解析，如果这个开端是正确的话。你只能一条一条地将指令解析出来，无法保证下一条指令到底有多长，有时候你甚至无法保证下一段数据确实是指令。

所以当你拿到一段“可执行指令”，诸如“请取出第 57 条到第 68 条指令”之类的要求时，绝对没有令人期望的那么简单了，即使只打算取得第 57~68 这几条指令，你也只能从第一条指令开始解析。

因此修改已经存在于内存中的二进制代码是异常艰难的一件事。比如，我打算将某

个函数内的某条指令替换成所期望的其他指令，那么一个显然的前提就是我必须首先找到那条指令的字节位置，并且用来替换的内容和被替换者必须是同样长的。太长的话没有空间可以放下，我们无法移动后面的指令；太短的话，不加特殊处理会导致前面所说的逐条指令解析过程崩溃。程序也无法被正常地执行下去。

nop（空操作）指令是单字节，机器码为 90h，这个字节可以用来填充多余区域。如果短指令替换了长指令，后面多余字节可以用 nop 填充。

但是这个过程的限制实在是太多了，除非是打算把 mov eax,1 改为 mov eax,0 这样简单的小事，否则根本无法通过简单的替换几条指令的方法做好，要替换更多的指令会带来更多的麻烦。

修改内存中的二进制代码的一个可行方案是：把少数几条指令拷贝到另外一个地方，然后在这些指令腾出的空间中写入 jmp 语句，跳到期待加入（或者修改为）的功能代码处。那些代码执行的结尾，再跳回前面所谓的“另一个地方”，把先前拷贝出来的几条指令执行掉，然后再跳回这些代码原本应该在的地方继续执行。（如果头晕了，请把这段话再读一次。）这是许多修改 Windows 内核加入自己的“补丁”的一种办法。

这不是什么很光彩的办法，看起来更像破解者（cracker）或者病毒。实际上这些技术可以很好地用于破解和病毒，不过令人惊奇的是，安全软件、一些内核工具也大量使用类似的方法，同时微软自己发布的应用层的 APIHook 也用类似的方法。

在 Vista 的 64 位版本下，部分内核代码受到 patchguard 的保护，这样做就有一些限制了。虽然相关的破解文章已经发布在网上，而且 patchguard 技术也受到了安全厂商的广泛抗议。

在 Vista 之前的 Windows 版本以及 Vista 的 32 位版本中，这些技术还大有用武之地。在以后的版本中，在安全厂商和微软以及破坏者的无休止的斗争过程中，可能会有新的替代技术产生。无论如何，充分了解 Intel 的机器码，对我们来说不会是浪费时间的一件事，让我们来认识这些 0 和 1 组成的东西吧。

11.1.2 单条指令的组成

首先是一条指令的组成，大致如下所示：每一格是一个组成部分，这些组成部分每个都是一个或者数个字节。它们中间又分成一些位，比如 MOD-REG-R/M 实际上分成了 MOD（2 个位）、REG（3 个位）和 R/M（3 个位）。请不要被这些奇怪的组成名吓倒。

前缀 （可选）	操作码	MOD-REG-R/M （可选）	SIB （可选）	地址偏移 （可选）	立即数 （可选）

前缀：指令前缀，最多可能有 4 个前缀。一条指令可以有 0~4 个前缀，但不允许一个前缀重复 2 次。

前缀又可分为 3 种：普通前缀（Prefixes）、指示性前缀（Mandatory Prefix）和 64 位扩展前缀（REX Prefix）。下面介绍普通前缀，其中又分为重复前缀、锁前缀和几种超越前缀。

- 重复前缀、锁前缀
  - f0h：Lock（锁定）
  - f2h：REPNE/REPNZ（当 0 标记为 1 且 rcx!=0 时重复）
  - f3h：REP/REPE/REPZ（当 0 标记为 0 且 rcx!=0 时重复）

锁前缀是用来锁定总线的，然后进行写操作，它主要用于自旋锁（spin locks）或其他需要进行多 CPU 同步处理的场合。只有很少一部分指令支持锁前缀，并且该指令第一个操作数必须为内存间接访问，否则将导致处理器异常。

重复前缀被认为是和锁前缀同样的类型，所以一条指令前不能同时包括重复前缀和锁前缀，不过那只是 Intel 的说法，据说 AMD 好像把它们当做不同的类型处理。重复前缀用在重复操作的场合比如字符串操作、IO 串操作（ins, outs）等，默认以(r)cx 作为计数器。

这里要注意的是重复前缀也可以被作为指示性前缀而用于 SSE 指令。

- 寄存器超越前缀和地址超越前缀

66h 和 67h：66h 称为“寄存器超越前缀”；67h 称为“地址超越前缀”。如果在 32 位模式下使用 16 位的寄存器，那么，这条指令将出现 66h 前缀；如果在 32 位模式下使用 16 位的地址，那么，将出现 67h 前缀；如果两个前缀同时出现，那么 67h 总是在 66h 前面。这两个前缀可以只出现一个。

如果在 16 位模式下，这两个前缀的出现则意味着相应地使用 32 位的寄存器长度和地址长度。总而言之，这两个前缀意味着 32 位模式下使用 16 位地址或寄存器，或者 16 位模式下使用 32 位地址或寄存器。如果没有这种“切换”，则不需要这两个前缀。

这里忽略了 64 位情况，请结合下面的 64 位扩展前缀再进一步了解。

- 64 位扩展前缀（REX 前缀）

REX 前缀非常重要，尤其是在下一节介绍 MOD-REG-R/M 扩展时。REX 只有在64位模式下有效，这个前缀必须紧靠操作码，不然将被忽略。相对于指示性前缀，REX 前缀具有更高的优先级。REX 的范围是 40h~4Fh。不过这带来了一个问题，就是原来 16、32 位下的 inc、dec 这些指令的操作码刚好在 40h~4fh 范围内。这样一来就发生了冲突，这意味着 64 位下的 inc、dec 指令的操作码将必须改变。

REX 前缀的一个字节内部区分如下：

7~4 位	3 位	2 位	1 位	0 位
REX，必须为 0100b	W	R	X	B

W 代表 Width，意味着宽度。在以后的描述中，本书称之为 REX.W，其他的位也类似。下面是 REX.W 和 66h 前缀，对 64 位模式下指令操作数长度的影响。

REX.W	有 66h 前缀的情况	默认操作数长度
1	被忽略	64 位
0	16 位	32 位
0	32 位	16 位

- R 代表 Register，寄存器，用于扩展 MOD-REG-R/M 位。
- X 代表 indeX，用于扩展 SIB 索引字节。
- B 代表 Base，用于扩展 MOD-REG-R/M 位或者 SIB 基域。

这些内容在后面遇到的时候再做具体的解释。

操作码：操作码可能有 1~2 个字节。大部分的操作码在一个字节内，依然分成了几个部分，说明如下（但是要注意，并不是全部操作码都是如此）：

7~2 位，操作码	D	W

了解 D 和 W 有助于理解后面更让人头晕的 MOD-REG-R/M 字节。D 和 W 描述着这样一种情况：许多指令在数据之间传输，大家都知道数据不可能直接从内存直接传送到内存，要么从寄存器传送到寄存器，要么从内存传送到寄存器，要么从寄存器传送到内存。那么，在这样的一条指令中，需要说明的就是，数据从哪里传送到哪里，以及传送多少数据。

CPU 的工作模式（32 位还是 16 位）以及前面说过的 66h 前缀已经决定了寄存器长

度。也就是说，传送多少数据是由它们决定的，但是仅限 W=1 的情况：如果 W=0，则无论前面指定了哪种模式，传送的都是 1 个字节。

W：为 1 时，传送的数据为字或者双字（16 或者 32），取决于 66h 前缀和工作模式（16 或者 32）；为 0 时，传送的数据为 1 个字节。

D：传输的方向。在 Intel 的指令中，总是用“REG（寄存器）”和“R/M（寄存器或者内存地址）”这样的两个参数来描述数据从源传递到目的的过程。REG 总是表示一个寄存器，R/M 则可以表示一个寄存器或者一个内存地址。当 D=1 的时候，数据将从 R/M 传输到 REG；当 D=0 的时候，数据将从 REG 流到 R/M。详见下面的 MOD-REG-R/M 字节。

11.1.3　MOD-REG-R/M 的组成

MOD-REG-R/M 是可选的，指令不一定有这项。是否存在这个字节仅仅取决于指令的定义，因此只能直接从前面的操作码来判断是否存在 MOD-REG-RM 字节。从操作码的本身我们已经可以知道一些操作数的类型信息，但也许还需要更多的信息，而这个字节的作用则是确定指令是否需要 SIB 字节、地址偏移和立即操作数，除此之外还定义了使用的寄存器。

MOD-REG-R/M 一个字节被分成了 3 个部分。MOD（2 个位）、REG（3 个位）和 R/M（3 个位）。REG 有 3 个位，总是用来表示寄存器。R/M 的 3 个位则有时候用来表示寄存器（R），有时候用来表示内存地址（M），而具体的表现，则取决于 MOD（模式）。这个字节如下：

7~6 位 MOD	5~3 位 REG	2~0 位 R/M

MOD：MOD 字段有两个位，那么，只有 4 种可能，也就是只有 4 种意义，见表 11-1。

表 11-1

MOD 的值	意　义
00	没有使用地址偏移
01	使用了 8 位的地址偏移
10	使用了 16 位或者 32 位的地址偏移。由当前模式决定 16 位或 32 位，当然默认值也可以被超越前缀所修改
11	仅仅使用了通用寄存器。此时 R/M 一定是寄存器，此外的情况是，R/M 是地址

REG-R/M：当 R/M 表示寄存器的时候，和 REG 部分的表示方法相同。因为都是3个位，表示 0~7 的时候分别表示了 8 种寄存器，见表 11-2。

表 11-2

REG 或 R/M 值	字节	字	双字	SSE	MMX
000	al	ax	eax	ss0	mm0
001	cl	cx	ecx	ss1	mm1
010	dl	dx	edx	ss2	mm2
011	bl	bx	ebx	ss3	mm3
100	ah	sp	esp	ss4	mm4
101	ch	bp	ebp	ss5	mm5
110	dh	si	esi	ss6	mm6
111	ah	di	edi	ss7	mm7

考虑 64 位扩展前缀时的情况。REX.R 是 1，那么 MOD-REG-R/M 的 REG 域扩展成了 4 位，从而可以索引 16 个寄存器。同样，如果 REX.B 是 1，那么 MOD-REG-R/M 的 R/M 域也扩展成了 4 位，同样可以索引 16 个寄存器。除了前述的寄存器之外，还可以增加 r8~r15 这 8 个新的通用寄存器。

当 R/M 表示内存地址的时候，需要知道的是表示哪个地址。请注意，在 Windows 下，这是一个线性地址（虚拟地址和线性地址同）。至于线性地址和物理地址之间的变换，在第 12 章中将会详细介绍。下面仅仅列出 R/M 在 32 位时的寻址方式，你会看到，R/M 总是将内存地址存储在寄存器中，见表 11-3。

表 11-3

R/M 的值	表示的地址
000b	ds:[eax]
001b	ds:[ecx]
010b	ds:[edx]
011b	ds:[ebx]
100b	使用 SIB 字节
101b	ss:[ebp] 使用特殊寻址
110b	ds:[esi]
111b	ds:[esi]

当 R/M 为 100b 的时候，表示还需要 SIB 字节。SIB 字节以及这种寻址方式将在下

一节中介绍。

当 MOD 为 00 并且 R/M 为 5（表示 ebp）时，表示 MOD-REG-R/M 字节后还有地址偏移。这也是为什么用 ebp 间接寻址的时候比用其他寄存器要多占用一个字节[ebp+00]的原因。同时当 R/M 为 5 的时候，也可以使指令直接访问内存地址。例如：

```
mov [0x12345678], ebx
```

否则，单靠 MOD-REG-R/M 是无法实现直接内存访问的。

在 64 位模式下，情况有所不同。上面的情况将变为：

```
mov [rip+0x12345678], ebx
```

在这种情况下，rip 将成为基址寄存器，而如果要直接访问内存的话，则需要借助于SIB 字节，甚至你可以用来获得当前执行指令的地址 lea rax, [rip+0]，虽然在 32 位下是不允许的。

11.1.4　其他的组成部分

基础知识

这里所谓的其他部分包括 3 个组成部分：SIB、地址偏移和立即操作数。

首先是 SIB 字节。SIB 的意义是 Scale-Index-Base，占用一个字节，如下所示。

7~6 位	5~3 位	2~0 位
Scale	Index	Base

SIB 也是可选的，如果存在的话，必须跟在 MOD-REG-R/M 的后面，用于 R/M 部分使用比例变址寻址方式时。比例变址时，SIB 可以表示一个地址，这个地址的计算方法为：

```
地址 = Base(表示寄存器内容)+ Index(表示寄存器内容)* (2 ^ Scale)
```

上面的（2 ^ Scale）表示 2 的 Scale 次方。因为 Scale 是两个位，因此这个数字只可能是 1、2、4、8 四种可能。Index 和 Base 各表示一个通用寄存器的当前值，表示的寄存器和 REG、R/M 表示寄存器的方法完全相同。此外，当 REX.X 为 1 的时候，Index 被扩展为 4 个位；REX.B 为 1 的时候，Base 被扩展为 4 个位，此时它们可以表示 16 个通用寄存器。

然后是地址偏移字节，同样是可选的，是否使用依赖于前面的 MOD-REG-R/M，可能为 1、2、4 字节。如果存在，则必须位于 MOD-REG-R/M 字节或者 SIB 字节的后面。

最后是立即数字节，取决于寻址方式的需要，为有符号数，可能为 1、2、4 字节。（取决于工作模式以及是否有操作数超越前缀。）

思考与练习

下面通过一个练习来理解并总结本节内容。请写出在 32 位模式下工作的 CPU 的以下汇编对应的机器码：

```
mov eax, dword ptr [ebx+4*ecx]
```

解答如下：首先是 32 位模式，并且操作数和地址都是 32 位的，所以不需要 66h 和 67h 前缀。第一个字节应该是指令码 8bh。

然后是 MOD-REG-R/M 字节。MOD 应该是 00b，因为没有使用地址偏移，此外也不是全寄存器操作。REG 应该是代表 eax 的 000b。RM 是比例变址寻址，那么应该是 100b。MOD-REG-R/M 总体为 00000100b，也就是 04h。

由于需要 SIB，所以最后需要一个 SIB 字节。其中 Scale 显然为 10b，这是因为 4 是 2 的 2 次方。Index 表示了 ecx，ecx 的表示方法为 001b。Base 表示了 ebx，ebx 的表示方法为 011b，总而言之，SIB 字节为 10001011b，改写为十六进制数字为 8bh。

那么，一共 3 个字节的机器码翻译正确结果为：8b048bh。

知识小结

从这里看，通过人工从汇编语言翻译为机器指令是可行的。但是，大量指令的汇编非常的烦琐，应该使用汇编编译器来做这件事情。

反汇编引擎则做与之相反的工作，把机器指令翻译成能够理解的汇编指令。翻译结果并不限于人们阅读的字符串，而是含有指令各种信息的数据结构。

下面本书将展示一个反汇编引擎的实现过程。

11.2　反汇编引擎 XDE32 基本数据结构

基础知识

当需要在程序中动态分析指令的时候，反汇编引擎是必要的代码。一个反汇编引擎的作用是：把机器码解析成可以理解的指令。这个理解不仅仅针对人，也可以针对程序。这样我们就不必开发严格依赖于具体实现的代码，而可以根据读取的机器码的分析结果进行随机应变了。而且幸运的是，反汇编引擎通常很小，网上能找到很多开源的反汇编引擎，加入到你自己的程序中并不困难。

反汇编引擎的目标就是把这些解析出来，我使用了 XDE32 反汇编引擎。XDE32 是开源反汇编引擎，主要由 3 个文件组成，即 xde32_table.h、xde32.h 和 xde32.c，有兴趣的读者请自己下载研究。

请注意 XDE32 是一个纯 32 位的反汇编引擎，因此不能识别 64 位指令。

代码分析

XDE32 用一个结构来代表一条指令的所有信息，这个结构如下：

```
#pragma pack(push)
#pragma pack(1)
struct xde_instr
{
        unsigned char       defaddr;     // 2 或者 4，地址字节数
        unsigned char       defdata;     // 2 或者 4，数据字节数
        unsigned long       len;         // 这个结构对应的指令的
                                         // 总长度
        unsigned long   flag;            // 指令特征标记
        unsigned long   addrsize;        // 地址偏移量的字节数（1,2,4）
        unsigned long   datasize;        // 立即数的字节数（1,2,4）
        unsigned char   p_lock;          // 是否有加锁前缀，0 或 F0
        unsigned char   p_66;            // 是否有数据前缀，0 或 66
        unsigned char   p_67;            // 是否有地址前缀，0 或 67
        unsigned char   p_rep;           // 是否有重复前缀 0，F2 或 F3
        unsigned char   p_seg;           // 是否有段地址寄存器覆盖
                                         // 前缀 0 或 26/2E/36/3E/64/65
```

```
    unsigned char       opcode;       // 操作码，当操作码为 0F 的
                                      // 时候，有指令第二字节
    unsigned char    opcode2;         // 指令第二字节
    unsigned char       modrm;        // 指令中的 ModR/M 字节
    unsigned char       sib;          // 指令中的 SIB 字节
    unsigned long    src_set;         // 指令操作源对象（包括寄
                                      // 存器、存储器等）
    unsigned long    dst_set;         // 指令操作目标对象（包括寄
                                      // 存器、存储器等）
    union                             // 地址
    {
        unsigned char       addr_b[8];
        unsigned short   addr_w[4];
        unsigned long       addr_d[2];
        signed char      addr_c[8];
        signed short     addr_s[4];
        signed long      addr_l[2];
    };
    union                          // 立即数
    {
        unsigned char       data_b[8];
        unsigned short      data_w[4];
        unsigned long       data_d[2];
        signed char      data_c[8];
        signed short     data_s[4];
        signed long      data_l[2];
    };
}; /* struct xde_instr */
#pragma pack(pop)
```

以上信息就覆盖了一条指令的所有信息。作为一个反汇编引擎，必须解析一个机器码并把所有的信息填入其中。一个指令对应的指令特征起非常大的作用。在 XDE32 中，特征是自己定义的，用来描述一个指令的结构、各段的字节长度等信息。每个操作码都拥有一些指令特征，这决定了下面如何进行继续解析。所有的指令特征可以列成一个属性表，定义如下：

```
#define C_ADDR1      0x00000001   // }
#define C_ADDR2      0x00000002   // } 指令的地址字节的有效位
#define C_ADDR4      0x00000004   // }
#define C_MODRM   0x00000008      // 指令有一个 ModR/M 字节
#define C_SIB        0x00000010   // 指令有一个 SIB 字节
#define C_ADDR67     0x00000020   // 地址长度覆盖前缀存在，
                                  // 此时地址字节数为 defaddr
```

```
#define C_DATA66     0x00000040          // 立即数长度覆盖前缀存在
                                         // 此时立即数字节数为 defaddr
#define C_UNDEF      0x00000080          // 寄存器没有定义（这条指令
                                         // 操作未知的寄存器）
#define C_DATA1      0x00000100          // }
#define C_DATA2      0x00000200          // }指令的立即数字节的有效位
#define C_DATA4      0x00000400          // }
#define C_BAD        0x00000800          // 坏指令，很少用到的一个标记
#define C_REL        0x00001000          // 这是跳转指令 jxx 或者 call
#define C_STOP       0x00002000          // 这是回跳指令，ret 或者 jmp
#define C_OPSZ8      0x00004000          // 操作数的大小是 8 位，如果没
                                         // 有这个标记则是 16 位或者 32
                                         // 位
#define C_SRC_FL     0x00008000          // 这条指令需要读取标志
                                         // 寄存器
#define C_DST_FL     0x00010000          // 这条指令影响标志寄存器
#define C_MOD_FL     (C_SRC_FL+C_DST_FL) // 上面两条的组合
#define C_SRC_REG  0x00020000// 这条指令需要读取通用寄
                                         // 存器
#define C_DST_REG  0x00080000// 这条指令需要写通用寄存
                                         // 器
#define C_MOD_REG  (C_SRC_REG+C_DST_REG)  //上面两条的
                                         // 组合
#define C_SRC_RM   0x00040000            // 读取 R/M 的数据
#define C_DST_RM     0x00100000          // 写 R/M 的数据
… …
```

以上是从代码中截取的部分指令特征。读者可以看到，表里面用不同的位代表不同的特征，当一条指令有多种特征的时候，可以将多个位组合起来。

下面的表正是指令操作码的特征表，这里，按操作码从 0x00 开始排列，每一行代表一个操作码对应的指令特征的集合，这是一个表，在头文件 xde32_table.h 中。

```
unsigned long xde_table[ TBL_max ] =
{
        // add modrm
/* 00 */
  C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM+C_OPSZ8,
/* 01 */
  C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM,
/* 02 */
 C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM+C_OPSZ8,
/* 03 */  C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM,
        // add al, c8
```

```
/* 04 */  C_DATA1+C_DST_FL+C_MOD_ACC+C_OPSZ8,
          // add ax/eax, c16/32
/* 05 */  C_DATA66+C_DST_FL+C_MOD_ACC,
          // push es
/* 06 */  C_BAD+C_PUSH+C_SPECIAL,
          // pop es
/* 07 */  C_BAD+C_POP+C_SPECIAL,
          // or modrm
/* 08 */
 C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM+C_OPSZ8,
/* 09 */  C_MODRM+C_DST_FL+C_SRC_REG+C_MOD_RM,
/* 0A */
  C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM+C_OPSZ8,
/* 0B */  C_MODRM+C_DST_FL+C_MOD_REG+C_SRC_RM,
          // or al, c8
/* 0C */  C_DATA1+C_DST_FL+C_MOD_ACC+C_OPSZ8,
          // or ax/eax, c16/32
/* 0D */  C_DATA66+C_DST_FL+C_MOD_ACC
… …
}
```

这个表很长，有多少条指令就有多少行。为了节约篇幅，我只引用了前面的小部分。从这个表里可以迅速找到一个指令的特征用于之后的分析，这个表是反汇编引擎的核心所在，读者可以在网上搜索到完整的表。

11.3 反汇编引擎 XDE32 具体实现

这一节来看看 xde32.c 中的实现部分。首先是汇编函数：

```
int __cdecl xde_asm(/* OUT */ unsigned char* opcode,
                    /* IN */ struct xde_instr* diza);
```

这个函数输入为前面的 xde_instr 结构，也就是被解析过的指令；输出为指令机器码。读者会发现用它生成机器码真的是很方便的一件事情：

```
int __cdecl xde_asm(/* OUT */ unsigned char* opcode,
                    /* IN */ struct xde_instr* diza)
{
  unsigned char* p;
  unsigned int i;
  p = opcode;
```

```
    // 首先写各个前缀
    if (diza->p_seg )              *p++ = diza->p_seg;
    if (diza->p_lock)              *p++ = diza->p_lock;
    if (diza->p_rep )              *p++ = diza->p_rep;
    if (diza->p_67  )              *p++ = diza->p_67;
    if (diza->p_66  )              *p++ = diza->p_66;
    // 然后写操作码
    *p++ = diza->opcode;
    if (diza->opcode == 0x0F)       *p++ = diza->opcode2;
    // 写 ModR/M 字节
    if (diza->flag & C_MODRM)       *p++ = diza->modrm;
    if (diza->flag & C_SIB)         *p++ = diza->sib;
    // 写地址字节
    for(i=0; i<diza->addrsize; i++) *p++ = diza->addr_b[i];
    // 写立即数字节
    for(i=0; i<diza->datasize; i++) *p++ = diza->data_b[i];
    // 返回写了多少个字节
    return p - opcode;
}
```

当然这离一个汇编编译程序还差得远，一个汇编编译程序应该能解析字符串，根据汇编助记符来生成机器码，绝对不是通过解析过的机器码信息。可见反汇编引擎和汇编编译程序的差别还是很大的。下面看反汇编部分的实现，这个函数的原型是这样的：

```
int __cdecl xde_disasm(/* IN */ unsigned char *opcode,
                       /* OUT */ struct xde_instr *diza);
```

我们拿到的是机器码 opcode（只知道所在指针，不知道到底有多长），i386 的指令是变长指令，你不得不去把一堆连续的字节拆解成一条一条的指令，而且还没有非常直观的方法。这个函数返回的是当前解析出的这条指令有多长，这样，你就可以把指针移动一个长度来解析下一条指令，解析出的信息放在 diza 中。下面是具体的实现过程：

```
int __cdecl xde_disasm(/* IN */ unsigned char *opcode,
                       /* OUT */ struct xde_instr *diza)
{
    unsigned char c, *p;
    unsigned long flag, a, d, i, xset;
    unsigned long mod, reg, rm, index, base;
    // 操作码
    p = opcode;
    // 把 diza 清 0
    memset(diza, 0x00, sizeof(struct xde_instr));
    // 得到数据字节数和地址字节数
    diza->defdata = XDE32_DEFAULT_ADDR/8;
```

```
    diza->defaddr = XDE32_DEFAULT_DATA/8;

    // 指令的特征标记，先清 0
    flag = 0;

    // 如果前两个字节为全 0 或者全 F，认为是坏指令
    if (*(unsigned short*)p == 0x0000) flag |= C_BAD;
    if (*(unsigned short*)p == 0xFFFF) flag |= C_BAD;
    … …
}
```

前面只是一些准备活动，下面的解析按前面的指令分解，分为这六步：

第一步，解析前缀。

第二步，解析操作码，并取得对应的指令特征。

第三步，根据指令特征，如果存在 MOD-REG-R/M 字节，则解析它。

第四步，根据前面的解析，如果存在 SIB 字节，则解析它。

第五步，根据前面的解析，如果有地址字节存在，则解析它。

第六步，如果有立即数存在，则解析它。

中间每一步中还会随时统计涉及的（被读或者被写）寄存器或者存储器等。这个统计比较烦琐，几乎占用了中间绝大多数篇幅。读者对此不需要很感兴趣，因此这里就省去了。

```
int __cdecl xde_disasm(/* IN */ unsigned char *opcode,
                       /* OUT */ struct xde_instr *diza)
{
……
        // 解析前缀
        while(1)
        {
    c = *p++;
    // 数据覆盖前缀
    if (c == 0x66)
        {
          diza->p_66 = 0x66;
          diza->defdata = (XDE32_DEFAULT_DATA^32^16)/8;
          continue;
        }
        // 地址覆盖前缀…
```

```
        if(c == 0x67)
        {
          … …
}
// 各种重复前缀
if ((c == 0x26) || (c == 0x2E) || (c == 0x36) || (c == 0x3E) ||
    (c == 0x64) || (c == 0x65))
{
  … …
}
    } // 前缀解析到此为止
    // 前缀解析完后，c 就变成了操作码，取得指令特征
    flag |= xde_table[ TBL_NORMAL + c ];
    if (flag == C_ERROR) return 0;
    // 如果是 2 字节指令
    if (c == 0x0F)
    {
      c = *p++;
      // 根据第二字节取得指令特征
      flag |= xde_table[ TBL_0F + c ];
      if (flag == C_ERROR) return 0;
      diza->opcode2 = c;
      … … // 这里根据各种指令来获取影响到的寄存器和存储器
    }
    // 如果根据特征，有 MOD-REG-R/M 字节，则解析之
    if (flag & C_MODRM)
    {
      c = *p++;
    diza->modrm = c;
  }
  … …
    … … // 其他的步骤基本类似，这里一并略去
    … …
    return diza->len;  // 最后返回指令的总字节长度
}
```

看到这里，读者应该基本理解了一个反汇编引擎的工作原理。限于作者的水平，没有详细地介绍，但是相对于下面要用的技术，已经足够了。若有兴趣，读者可以自己开发一个反汇编引擎，不过这是一个艰巨的工作，虽然往往需要的代码并不是很多，32 位的反汇编引擎有很多，但是 64 位的 x86 反汇编引擎很难找到。也许最有效的方法，还是对着 Intel 的指令手册自己写一个。

第 12 章　CPU 权限级与分页机制

12.1 Ring0 和 Ring3 权限级

基础知识

本节的内容以知识为主，比较少技巧和经验。读者只需要了解，不需要熟练。如果你熟悉 x86 架构，请直接跳过这节。

现在探讨内核程序和应用程序之间的本质区别。除了能用 WDK 编写内核程序和阅读一部分 Windows 的内核代码之外，我们还需要了解它们的本质是什么，它们和我们熟悉的应用程序有什么区别。

Intel 的 x86 处理器是通过 Ring 级别来进行访问控制的，级别共分 4 层，从 Ring0 到 Ring3（后面简称 R0、R1、R2、R3）。R0 层拥有最高的权限，R3 层拥有最低的权限。按照 Intel 原有的构想，应用程序工作在 R3 层，只能访问 R3 层的数据；操作系统工作在 R0 层，可以访问所有层的数据；而其他驱动程序位于 R1、R2 层，每一层只能访问本层以及权限更低层的数据。

这应该是很好的设计，这样操作系统工作在最核心层，没有其他代码可以修改它；其他驱动程序工作在 R1、R2 层，有要求则向 R0 层调用，这样可以有效保障操作系统的安全性。但现在的 OS，包括 Windows 和 Linux 都没有采用 4 层权限，而只是使用 2 层——R0 层和 R3 层，分别来存放操作系统数据和应用程序数据，从而导致一旦驱动加载了，就运行在 R0 层，就拥有了和操作系统同样的权限，可以做任何事情，而所谓的 rootkit 也就随之而生了。

rootkit 在字面上来理解，是拥有“根权限”的工具。实际上，所有的内核代码都拥有根权限，当然，并不一定它们都叫做 rootkit，这要看你用它来做什么。用 rootkit 技术开发的木马和病毒正在迅速发展，它们往往极难清除，以往杀毒软件可以轻松清除掉系统中病毒的时代似乎已经一去不复返了。

基础知识

大多数指令可以同时使用于 R0 层和 R3 层，但有些和系统设置相关的指令却只能在 R0 层被使用，或者在 R3 层的使用受到限制，主要有下面这些：

- lgdt：加载 GDT 寄存器
- lldt：加载 LDT 寄存器
- ltr：加载任务寄存器
- lidt：加载 IDT 寄存器
- mov：加载和存储控制寄存器、调试寄存器时受限
- lmsw：加载机器状态字
- clts：清除 cr0 中的任务切换标记
- invd：缓冲无效，并不写回
- wbinvd：缓冲无效，并写回
- invlpg：无效 TLB 入口
- hlt：停止处理器
- rdmsr：读模式指定寄存器
- wrmsr：写模式指定寄存器
- rdpmc：读取性能监控计数器
- rdtsc：读取时间戳计数器

最后 2 条指令 rdpmc 和 rdtsc，在 cr4 的位 4（PCE）和位 2（TSD）被设置的情况下可以同时被 R0 层和 R3 层调用。任何违反上面规定的操作，在 Windows 下都可能会产生通用保护故障的异常。

另外，还有些所谓的 IO 敏感指令，包括：

- cli：关闭中断
- sti：开启中断
- in：从硬件端口读
- out：往硬件端口写

这些指令在 R0 层可以直接被使用，在 R3 层被使用的时候还要检查 IO 许可位图，综合判断是否允许调用。

当然，前面已经声明我们写的和研究的代码都是内核代码，也就是说，上面这些指令都是可以用的。当然，相应的 rootkit 技术的病毒和木马的作者显然也会明白这一点，所以这并不是让人很有安全感的一个现状。

更重要的保护机制是如何保证系统内存空间的读/写、可执行属性，这将在 12.2 节“保护模式下的分页内存保护”中详述。对于病毒和木马来说，使用硬件机制来实现破坏虽然并非不可能，但是远不如直接修改内存中的操作系统内核和其他软件的代码来得简洁方便，那是破坏与安全对抗的主战场。

12.2 保护模式下的分页内存保护

基础知识

x86 处理器的内存保护，分为分段保护和分页保护。

首先介绍 CPU 的运行模式，目前的 x86 CPU 的运行模式主要有以下几种。

- 实模式：CPU 在启动后马上处于实模式。大家熟悉的 DOS 就是运行在这种模式下，缺点是默认只能访问 1MB 内存，单任务；优点就是所有程序都运行在 R0 层，应用程序和系统程序具有同样的权限，可以说是病毒编写者的最爱。
- 保护模式：建议的 CPU 运行模式，支持分段、分页，可以访问更大的内存，可以运行多任务，大家熟悉的 Windows、Linux 均运行在该模式下。该模式提供了丰富的系统管理特性，同时也使现代病毒、木马与安全软件都有了更多的技术含量。
- 系统管理模式（SMM）：这个模式提供给操作系统或执行环境一个透明的机制来处理电源管理和 OEM 独有的特性。
- 虚拟 8086 模式：为了能在保护模式中运行原来能在实模式下运行的程序，所特意提供的一种虚拟的 8086 模式，在这种模式下允许在保护模式的多任务环境下执行实模式程序，所以我们可以在 Windows 9x、Windows 2000 下运行大多数的 DOS 程序。
- 64 位扩展模式：Intel 和 AMD 均推出了支持 64 位的 CPU，支持 64 位的线性寻址，其又分为如下几个模式：

- 64 位模式：支持 64 位线性寻址，支持超过 64GB 的物理地址，所有 64 位程序均运行在此模式下。
- 兼容模式：原有的 32 位程序运行于此模式下，从而可以使 32 位程序不用修改就直接运行于 64 位操作系统下，部分避免了曾重金购买 32 位软件的客户的怒火。

x86 处理器的地址分段模式包括以下几种形式。

- 平坦模式：在这种模式下，系统能访问一个连续的、不分段的地址空间，所有的段被映射到同一个地址空间，所有的段都有相同的基地址 0，界限为 4GB（在 32 位情况下）。我们所用的操作系统，无论是 Windows 还是 Linux 都运行在这种模式下。这种简洁、易懂的模式深受程序员的喜爱。
- 多段模式：不同的段，如代码段、数据段、堆栈段等位于不同的段中。我曾经花了很大的努力去理解段和偏移量，以及地址如何换算等，绞尽脑汁之后发现实际上 Windows 和 Linux 都用了平坦模式，所以我又忘记了它们。

重要细节

分页机制是 Windows 和 Linux 都利用的技术。我们理解分页机制的意义在于，理解反汇编时看到的“存储器地址”。比如如下的一句：

```
818c1526 call dword ptr [nt!_imp_KeAcquireQueuedSpinLock
(81801100)]
```

在这里我们看到了两个地址：第一个是 818c1526，这是本指令所在的存储器地址，其次是 81801100。

后者是符号 nt!_imp_KeAcquireQueuedSpinLock 所对应的地址，实际上就是函数 KeAcquireQueuedSpinLock 的入口地址。那么，这些地址有什么意义呢？

我们在 WinDbg 中看到的地址都是线性地址，其中含有 3 个部分：页目录、页和页中偏移。

这里使用了“分页”机制。分页机制的特点是，我们在调试器中看到的地址数据是一个“线性地址”（很多书上详尽介绍“虚拟地址”，但是在平坦段模式下，虚拟地址就是线性地址，所以我完全忽略段和选择子的概念。下面只讲述线性地址），这个地址需

要通过特殊的转换来修正为实际的物理地址。也有可能对应的物理地址根本不存在，这就引发所谓的“缺页中断”，操作系统会从硬盘交换一页到物理内存中，从而使这一页变成存在的。

在一个进程中，我们可以访问 0~4GB 的线性内存空间。在 Windows 中，这 0~4GB 被分成两个部分：0~2GB 范围内为应用程序进程独有的内存空间；2~4GB 范围内为系统内核地址空间。以下为一个进程看到的 0~4GB 线性地址空间的分布。

0~2GB 进程独有内存空间	2~4GB 系统内核地址空间

所以，我们在调试内核代码的时候，看到的地址大多是大于 80000000 的。2~4GB 的空间是完全共享的，每个应用程序进程看到的数据也都是一样的。如果予以修改，后果是严重的，这就是为何内核程序出错往往导致系统崩溃，而应用程序出错最多导致本进程非法退出的原因。

0~2GB 范围内的应用程序进程独有内存空间的意义是：你在这个进程内可以自由读/写这些空间，而其他进程看不到你的修改，这些地址内的数据是本进程独有的。要实现这种机制，必须让相同的线性地址映射到不同的物理地址。这是怎么实现的呢？

控制从线性地址到物理内存地址的转换的数据表被称为“页表”，这些页表也保存在内存中。系统中可以有多个页表，因此有一个能访问到每个页表的“页目录”，页目录的地址被保存在控制寄存器中。

一个 32 位的线性地址分为以下几个部分：

31~22 位 目录	21~12 位 页	11~0 位 偏移

第一个部分是页目录，第二个部分是页表项，第三个部分是偏移。从线性地址到物理地址的转换过程，如果忽略不必要的细节，是这样的：

① 线性地址中的目录加上 cr3 寄存器中的目录基地址得到页表项地址。

② 页表项地址加上 21~12 位的“页”得到了实际内存页的开始地址。

③ 开始地址加上偏移即可得到实际物理内存地址。

当然，你不能忘记缺页问题。实际上页表中保存了页的状态，如果该页不存在，对这个页的访问将触发缺页中断。

你可以注意到有 12 位来表示页中偏移，12 位能表示的数据显然不止 4KB，而是可以到 4MB 之多。实际上 CPU 是支持 4MB 分页的，只不过现在还没有软件支持。

现在问题很简单了：只要在切换不同的进程时，切换 cr3 中保存的目录基地址就可以了，每个进程独自维护不同的目录、页表体系即可。这样一来，虽然不同进程中看到一样的线性地址，但是物理地址可以完全不同，互相不干扰。

这就是 Windows 内存保护技术。总结起来主要有两点：R3 级的代码无法访问 2~4GB 范围内的内存；R3 级的某个应用程序进程中的代码，无法修改其他进程的 0~2GB 范围内的内存。

12.3 分页内存不可执行保护

12.3.1 不可执行保护原理

基础知识

前面提到，R3 级的代码无法访问 2~4GB 范围内的内存，但是还没有讲述这是如何做到的。前面提到页表项中有页的开始地址，实际上，一个页表项结构如下：

31~12 位 地址	11~6 位 保留	D	A	PCD	PWT	U 位	W 位	P 位

重要的是上面的 P 位：表示这个页是否存在；W 位：表示是否可写；U 位：是一个用户自定义属性，但是实际上，在 Windows 里如果这一位是 1，则表示这是用户页，可以被 R0 和 R3 级的代码访问；如果为 0，则是系统页，只能被 R0 级别的代码访问。

这里有一个问题，那就是页面只有读或者写的控制，并没有“是否可执行”的控制。实际上，你可以简单地用跳转指令，跳到某一个地址，CPU 就会继续在这里执行下去。这给许多病毒和木马带来可乘之机。

攻击代码最大的问题不在于如何进行破坏，而在于如何被执行。普通的用户不大可能直接执行恶意代码，但是他们可能执行合法的软件。这些软件在读入数据文件的时候，恶意代码可以被隐藏在这些数据中。当然，合法的软件理论上是不会将这些数据作为代价去执行的。

但是有漏洞的合法软件这时候可能被利用：通过缓冲区溢出，设法让这些软件开始执行本来只具有数据意义的（而不具有可执行意义的）“数据”，当然这些数据是精心构筑的。从程序的意义上来讲，这些数据应该位于栈或者堆中；从硬件的意义上讲，这些数据所在的页面应该禁止被执行，但是实际上，它们是可以被执行的。黑客们对它们的

利用乐此不疲。

AMD64 架构首先改变了这一境况，Intel 公司也迅速地跟进了。由于从 32 位扩展到了 64 位，所以前面的页表项也被扩展了。下面我们看到的是扩展后的页表项，有 64 个位，是以前的两倍。

NX	62~52 位 保留							51~32 位 地址
31~12 位 地址		D	A	PCD	PWT	U 位	W 位	P 位

其他的都不重要，但是增加了一个 NX 位（第 63 位），意义在于 No Excute（不可执行）。那么，只要这一位为 1，这个页面就不可以执行了。程序在分配自己的堆栈或者堆空间的页面的时候，就可以将这些位置为 1，这样即使有缓冲区溢出漏洞存在，也不大可能被利用来执行恶意代码。这就有效地封堵了这样一个漏洞。

不过糟糕的是，技术总是带着两面性。有一些安全方面的应用，典型的是应用程序的加壳：加壳是一种安全技术，将自己的代码预先加密，然后分配数据空间，解密这些代码，再放入分配的堆数据空间中运行，这样可以防止自己的重要信息被不受欢迎的读者阅读了解。其结果是，这些数据空间很可能不能执行，因为 NX 的保护。如果简单地将堆空间设置为可执行的，实际上又破坏了保护功能，实现加壳技术的难度大大地增加了。

Windows XP SP2 开始利用这样的特性，不过由此可能导致一些利用了上述特性的合法软件根本无法执行，所以 Windows 留下了关闭 NX 保护的开关。在 boot.ini 中，Windows 内核启动参数中增加/EXECUTE 即可关闭此功能，增加/NOEXECUTE 则启用此功能。

12.3.2 不可执行保护的漏洞

基础知识

遗憾的是，即使有了 NX 的保护，依然存在利用 Windows 漏洞的代码进行缓冲溢出攻击的可能。除去不支持 NX 的 CPU 外，启动了 NX 的 Windows 电脑依然存在漏洞。漏洞是，可以首先不要跳转到自己在堆中分配的代码，充分利用 Windows 的 NTDLL 中现存的代码（这些代码显然是可执行的，不受 NX 保护），利用这些代码关闭这个进程的 NX 保护，之后就可以为所欲为了。

常用的漏洞利用手段导致攻击者可以往堆栈中增加一个地址。我们前面已经学习过 ret 指令等于 pop 出一个地址，并跳到那个地址去执行。NX 保护开启后，如果这个地址

是一个栈空间或者堆空间中的地址，则导致异常。但是如果是一个 NTDLL，这样已经加载的可执行 DLL 中的地址则毫无问题。

代码分析

看下面的代码：

```
push dword ptr[esp]                          ;真实的返回地址入栈

;实际上这里在模拟一个函数入口，以便之后直接跳转到某些函数
;内部之后得以正确地返回。push esi 的次数取决于堆栈平衡性的需
;要
push ebp                                     ;前栈底备份
mov ebp,esp                                  ;当前栈底入 ebp
push esi
push esi
push esi                                     ;
push LdrpCheckNxCompatibility_0x13           ;伪造的返回地址
push NtDllOkeytoLockRoutine                  ;伪造的返回地址

ret
```

如果这两个地址（LdrpCheckNxCompatibility_0x13 和 NtDllOkeytoLockRoutine）被压入堆栈中，然后遇到函数正常返回，有什么后果呢？请打开 WinDbg，按我们前面学习的成果，输入：u NtDllOkeytoLockRoutine 并回车：

```
ntdll!NtdllOkayToLockRoutine:
7c952080 b001            mov     al,0x1
7c952082 c20400          ret     0x4
```

这里会把 al 置为 1，然后返回，并多弹出 4 个字节堆栈空间，返回的地址显然是我们先压入堆栈的 LdrpCheckNxCompatibility_0x13。此外多弹出 4 个字节的参数，刚好平衡了我们上面输入的倒数第一个 esi。

更精彩的是 LdrpCheckNxCompatibility_0x13 处，请继续跟踪：

```
ntdll!LdrpCheckNXCompatibility+0x13:
7c91d3f8 cmp     al,0x1          ;比较 al 是否为 1，是显然的
7c91d3fa push    0x2             ;压入一个数据
7c91d3fc pop     esi             ;弹出到 esi 中，此时 esi=2

;如果 al 是 1，则跳一个地址。让我们继续跟踪
7c91d3fdje  ntdll!LdrpCheckNXCompatibility+0x1a (7c93feba)
```

继续看 ntdll!LdrpCheckNXCompatibility+0x1a：

```
ntdll!LdrpCheckNXCompatibility+0x1a:
7c93feba mov     [ebp-0x4],esi        ;此时[ebp-0x4]=2
7c93febd jmp     ntdll!LdrpCheckNXCompatibility+0x1d (7c91d403)
```

然后是 ntdll!LdrpCheckNXCompatibility+0x1d：

```
7c91d403 cmp     dword ptr [ebp-0x4],0x0  ;不成立，下面跳转
7c91d407 jne  ntdll!LdrpCheckNXCompatibility+0x4d (7c935d6d)
```

经过一系列的跳转，我们终于能看到攻击者的真实意图了：

```
ntdll!LdrpCheckNXCompatibility+0x4d:
7c935d6d push    0x4                  ;倒数第一个参数为 0x4
7c935d6f lea     eax,[ebp-0x4]        ;倒数第二个参数为 0x2
7c935d72 push    eax
7c935d73 push    0x22                 ;倒数第一个为 0x22
7c935d75 push    0xff                 ;倒数第一个为-1

;下面是真实的意图：执行 ZwSetInformationProcess
7c935d77 call    ntdll!ZwSetInformationProcess (7c90e62d)
7c935d7c jmp  ntdll!LdrpCheckNXCompatibility+0x5c (7c91d441)

;在这里得以正常返回原来的返回地址
ntdll!LdrpCheckNXCompatibility+0x5c:
7c91d441 pop     esi
7c91d442 leave
7c91d443 ret     0x4
```

攻击者的意图是利用缓冲区溢出漏洞在堆栈里巧妙地构筑返回地址，使 CPU 最终执行 ZwSetInformationProcess，并得以正确地返回。ZwSetInformationProcess 按如下调用能关闭这个进程的 NX 保护。

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;
ZwSetInformationProcess(
    -1,                          //
    ProcessExecuteFlags,         // 0x22
    &ExecuteFlags,               // ptr to 0x2
    sizeof(ExecuteFlags));       // 0x4
```

以上代码执行之后，实际上这个进程的 NX 保护就被关闭了。现在并没有跳转到堆中或者栈的攻击代码中，只向堆栈中放了几个合适的数据，这段代码就被执行了，也就是说，攻击得到成功。当然这样的巧合是不容易找到的，因为首先需要找到 NtDllOkeytoLockRoutine 和 LdrpCheckNxCompatibility_0x13 的地址。但是攻击者无法搜

索：他不能执行自己写的代码，所以只能硬编码。但是硬编码的话，不同版本的 XP 都有可能不同，所以要保证攻击正常进行是非常困难的。但是这提醒了我们，危险依然是存在的。MS 很快会堵上这样的漏洞，然而新的漏洞又会被发掘出来。

上机实践

现在请读者上机实践，以熟悉在内核中的漏洞：虽然这并不一定是读者的工作，读者的工作可能是防止它们被黑客利用，但是如果不熟悉它们，那些都无从谈起。下面的 C 语言代码用来做测试。请注意这仅仅是模拟演示，这些代码不是在寻找某个软件构筑缓冲溢出攻击，而是试试上面的手段是否真的可以被使用，因此它们不会攻击任何软件，只改变自己进程的 NX 保护。要测试的话需要满足以下几个条件：

- 使用 Windows XP SP2 版本的操作系统。
- 使用支持 NX 的 CPU（一般 64 位的 AMD 和 Intel CPU 即可）。
- Windows XP 打开了 NOEXCUTE 支持。如果没有请修改 boot.ini 文件，启动参数后加上/NOEXCUTE。

下面的代码不是内核代码，读者随意建立一个 Windows 的应用程序工程即可测试。

示例代码

```
// 以下是两个关键地址。请千万不要照抄！为了防止被照抄
// 我特意把地址的部分改为了?号。正确的方法是打开 WinDbg
// 去调试得到目标机器上这两个地址。不同版本的 XP（甚至
// 语言版本不同），都可能得到不同的地址
#define NtdllOkayToLockRoutine            0x7c???????
#define LdrpCheckNXCompatibility_0x13     0x7c9??????

// 静态变量。你可以在其中写入代码，但是如果开启了 NX，试图
// 执行这些代码会导致异常
static unsigned char heap_code[0x100];

// 模拟的攻击代码
__declspec(naked) void attack_code()
{
    _asm
    {
        push dword ptr[esp]
```

```
        push ebp
        mov ebp,esp
        push esi
        push esi
        push esi
        push LdrpCheckNXCompatibility_0x13
        push NtdllOkayToLockRoutine
        ret
    }
}
```

请注意以上的代码，这些代码实际上是不会有的，这里只是用来生成堆栈里的模拟缓冲区溢出的数据而已。下面我们写一个主函数，执行这个主函数可以看出效果。

```
int _tmain(int argc, _TCHAR* argv[])
{
    heap_code [0]=0xc3;  //0xc3 是 ret 指令
    attack_code();
    _asm lea eax,heap_code
    _asm call eax
    return 0;
}
```

如果不执行 attack_code()，以上代码无法正常执行，触发异常，因为 heap_code 并不在可执行页上；但是，执行了 attack_code()之后，却可以正常执行。这就说明，巧妙地构筑攻击代码进行攻击来绕过 NX 是可行的。

思考与练习

安全软件的开发者该如何防止这种攻击方式呢？这个问题留给读者自己思考，并作为附加的练习。必须提醒的是，安全软件的权限远比得手之前的攻击者大：这些代码是可以随时执行的，而且运行在 R0 级，几乎可以做一切事情，而不用像攻击者那样必须利用那么巧妙的技术。

12.4 权限级别的切换

本节专门讲述权限级别切换的问题。这个问题是这样的：R0 是特权级，R3 是用户级别，如果一段代码执行在 R3 级别，无法切换到 R0 级别，那么这个问题很严重。因

为 CPU 一旦执行到 R3 级别的代码上，就无法回到 R0 了。显然，必须要能够从 R3 切换到 R0。

但是如果可以从 R3 切换到 R0，这又带来新的问题：如果位于 R3 的代码，可以自由地切换到 R0，那么，区分 R0 和 R3 的权限级别又有什么意义呢？这没有任何保护性，只是多了一道手续而已。本节的目的在于解释这样的疑惑。

12.4.1 调用门及其漏洞

在 x86 处理器中，cr0~cr4 为控制寄存器，要注意的是 cr0 的位 16，该位是 Intel 在 486 后引入的，在 486 之前对于只读页，只在 R3 层受控制，而对于 R0 层则无限制。在 486 以后引入了该位，因为很多 OS 的写时拷贝机制必须依赖在 R0 写只读页也要能引发页中断来实现，所以增加了该位来控制 R0 是否可以写只读页，如果为 0 则不能写，如果为 1 则可以写。而 R3 层的应用程序根本不能访问 cr0，这样就达到了既能实现 OS 机制，又不影响系统的安全性的目的。这一知识在下一节做内核 Hook 时会用到，请留意。

cr2 用来存放发生页中断时的线性地址，只在页中断处理程序中使用，页中断处理程序是操作系统的重要部分；cr3 中保留有页目录基地址；cr4 用来进行 CPU 的架构扩展。

系统中存在着一些系统表格，OS 通过它们进行管理和实现各种保护以及中断处理。比如常常被人提及的 GDT（全局描述符表）、LDT（局部描述符表）和 IDT（中断描述符表）。但是你不用关心 GDT、LDT 中的段描述符，因为在 Windows 中基本没有用到 LDT。GDT 中还保存着各种调用门和其他的描述符。

实际上，如果要通过中断门或者调用门，才能从 R3 进入 R0（在出现 sysenter 指令之前），而且进入的地址是填写在“门”记录中的。而这些门记录，都写在系统内存区（只有 R0 级别才能访问。请回忆前面的 12.2 节“保护模式下的分页内存保护”）。这样作为 R3 级别的代码，虽然可以切换到 R0 级别，但是只能切换到操作系统已经规定的那些地址。

这样一来，操作系统就可以事先填写一组调用门，并保证切换后的地址都是操作系统的“系统调用”，来防止普通用户随意切换到自己编写的 R0 级的代码。

在 Windows 9x 系统中，这个保护并不存在，任何 R3 级别的代码都可以直接添加新的调用门和中断门。有名的 CIH 病毒就直接利用了这样的技术执行了 R0 级的代码。

在 Windows NT/2000/XP 系列的操作系统中，这件事要更麻烦一些。下面我们来了

解调用门。

Windows 的调用门保存在 GDT 中。GDT 称为“全局描述符表”，其中不但有调用门，还有其他种类的描述符存在。在全局描述符表中，一个调用门描述符的结构如下图所示，请注意这里有两个 32 位数据，一共是 64 位。

31~16 位，段中偏移的 31~16 位	P	DPL	0	11~8 位，类型，必定为 1100b	7~5 位，全 0	4~0 位，参数计数
31~16 位 段选择子	15~0 位 段中偏移的 15~0 位					

在这里读者要确认一个概念，就是“段选择子”。段选择子所选择的是一个描述符，在这里只考虑全局描述符表的时候，可以认为一个段选择子就选择了一个 GDT 中的描述符。

有趣的是，调用门也是一个描述符，所以，下面会看到实际上是用另一个段选择子来找到这个调用门。

所有的全局描述符都存在全局描述符表中，结构和上面的调用门类似。其中 P 表示这个描述符是否有效，DPL 表示调用门的特权级，段选择子指出要访问的代码段，偏移量指出在代码段中的入口点，参数计数给出的是要传递的参数的数量（32 位下用双字的个数表示）。主程序通过堆栈把入口参数传递给子程序。如果利用调用门调用子程序时，引起特权级的转换和堆栈的改变，那么就需要将外层堆栈中的参数复制到内存堆栈，而复制的数量则由参数计数决定。

具体访问一个调用门的时候，必须在指令 call 或者 jmp 的操作数中提供一个指向调用门的远指针，来自于该指针的段选择子指定调用门。

这样的后果就是，如果黑客可以写入一个调用门，那么就可以利用 R3 程序直接进入 R0 权限。

全局描述符表的位置是很容易找到的。指令 sgdt 不是特权指令，在 R3 下直接调用就可以得到 GDT 的地址，然后目标就是搜索空闲位置插入一个调用门了。

难度在于如何写入 GDT 所在的内核地址区，R3 代码没有对该内存区域的访问权限。但是 Windows 内部提供了一个设备，名字叫做“Device\PhysicalMemory”，这个设备可以被系统用户访问，直接访问这个设备就可以访问物理内存。

从管理员（Administrator）得到系统（System）用户权限是可行的。在 Windows 的世界里，Administrator 实际上已经是至高权限了，只是有一些权限需要自己用多余的手续去获取。

然后就是物理内存和 GDT 的线性地址如何换算的问题。似乎有人为了研究这个问题反汇编了 MmGetPhysicalAddress 这个函数。这个动机是明显的，因为这个函数可以将线性地址转换为物理地址（但是只能在内核模式下调用），结果意外发现 80000000~a0000000 之间的地址转换极其简单，并没有查页表，只是简单的位移计算。

在这种情况下，写入调用门就完全可行了。以上信息我参考了网上流传的 WebCrazy 的《Windows NT/2000/XP 下不用驱动的 Ring0 代码实现》一文。这些技术虽然很有名，但是已经很老了。从 Windows 2003 SP1 开始，微软已经不允许用户程序（无论权限）写“Device\PhysicalMemory”了。此外我跟踪了 XP 最新版本下的 MmGetPhysicalAddress 函数，发现也和该文描述的不一样了。虽然我不肯定 80000000~a0000000 之间的地址转换方法是否改变了，但显然微软在填补这些漏洞。

但是我个人认为这不算是什么有价值的漏洞，因为以上的行为都需要管理员权限。管理员权限的用户，虽然是 R3 层，但已经足够去安装一个驱动程序了。直接调用 CreateService 等系列的 Win32API 函数就可以悄无声息地去安装一个内核程序，那就没有必要在 R3 层去获得 R0 权限了。事实上，现在流行的 90%的带有内核组件的木马和病毒都是这样做的。

有人问我：难道不用管理员用户病毒就无法安装驱动吗？

我的回答是：“除非是真正的 0Day 漏洞攻击，否则，根据我的知识，确实是这样的。”换句话说，这种攻击并不是真正的漏洞攻击。但是遗憾的是，依然给无数的用户带来了烦恼，因为 90%的个人用户都习惯用管理员用户使用自己的计算机，包括我自己。

最好的防范方法就是开发一个内核程序安装在计算机上，禁止写入新的调用门，并防止加载任何驱动程序（即使是管理员用户）。有必要安装时，弹出提示让用户确认，这样基本上可以挡住 90%的含有 rootkit 的病毒，除非病毒拥有未公开的 0Day 攻击手段，无法被拦截。许多的杀毒软件都有这方面的监控功能，学习第 13 章“开发 Windows 内核 Hook”之后，读者也可以 Hook 一些内核调用，来监控和阻止类似的行为。

瑞星的碎甲技术

瑞星曾经有一款“瑞星卡卡上网安全助手”软件，称含有独创的“碎甲技术”，能对抗大量的 rootkit 病毒。

不多久被反汇编牛人 doskey 将反汇编分析结果贴在驱动开发网。我没有读到那

篇分析的文章，但是“碎甲技术”的原理是很清楚的。通过禁止用户在系统中安装和加载不明的驱动程序，来防止含有 R0 级驱动程序的病毒的袭击。

实际上比较正式的安装驱动的接口都是公开的，只要 Hook 相关的内核调用进行检查就可以了。在后面的第 13 章中，读者会学到如何开发系统内核 Hook。

但是仅仅如此还不够。Windows 中不但有公开的安装驱动的接口，也有未公开的接口，于是一一挖掘出来就成了一件不容易的事情。除了目前已知的 ZwLoadDriver、ZwSetSystemInformation 等接口之外，传说中还有其他的方法可以加载驱动。同时，一定有人在日夜不停地搜寻新的漏洞。因此试图通过“碎甲技术”来获取绝对安全依然是不可能的。

在后面的第 14 章中，本书举出的实例也可以起到类似的效果，防止未认证驱动的加载（所不同的是除了驱动之外，R3 级别的可执行模块也控制了）。碎甲技术是一种缩小范围（仅限 R0 级驱动模块）的可执行模块认证技术。我相信可执行模块认证将成为基础的安全手段，也相信本书的每个读者都能够掌握它。

12.4.2 sysenter 和 sysexit 指令

从 R3 层切换到 R0 层，早期的 OS 一般都通过中断门来实现这个功能，比如 Linux 下 int 80h、Windows NT 和 Windows 2000 下的 int 2eh。中断门的用法和调用门差不多，而在 XP 以后则采用了快速系统调用 sysenter 和 sysexist 来进行系统调用和返回。

首先请明确以下两点：

- sysenter 和 sysexit 是机器指令，是在 PII 以后增加的，并非一个函数，或者与操作系统相关的系统调用。
- sysenter 在 R3 层下执行，跳转到 R0 层的代码；sysexit 在 R0 层下执行，回到 R3 层代码。

当然问题由此产生：如果说在 R3 下执行 sysenter 就可以跳转到 R0 级别，那么 R3~R0 的限制又有什么意义呢？如果我是病毒的作者，我就直接在 R3 级别调用 sysenter 来取得 R0 权限。

R3 层的代码可以调用 sysenter 跳转到 R0 层，但是却不能自由指定跳转到的目的地址。

问题在于：sysenter 虽然可以在 R3 下执行，但是跳转的目的地址却一定要在 R0 下指定。换句话说，我可以在 R3 下调用 sysenter，却无法控制我要跳转的地址。

跳转的地址是由 Windows 预先设置的，下面介绍这个地址的设置方法。这些地址保存在 3 个特殊的寄存器中，如表 12-1 所示。

表 12-1

寄存器	功　能	代　号
SYSENTER_CS_MSR	保存跳转之后的 cs 内容	174h
SYSENTER_CS_ESP	保存跳转之后的 esp 内容	175h
SYSENTER_CS_EIP	保存目标地址	176h

cs 是代码的段寄存器。cs 中保存的一个是一个选择子，选择一个 GDT 中的段描述符。Windows 中这个 cs 的设置很特别，这个在下面会介绍。R3 代码不能更改 cs。

跳转之后的 esp 的内容（堆栈指针）被保存在 SYSENTER_CS_ESP 中。一会看到，因为调用 sysenter 的代码一定是 R3 代码。R3 代码要使用自己的 esp 的话，临时去设置 SYSENTER_CS_ESP 显然不可行：没特权。获取 R3 的堆栈指针另有方法。请继续阅读下文。

跳转之后的 eip 是跳转之后执行的第一条指令的目标地址，这个被 Windows 固定死了。下面会看见，R3 代码无法更改它。设置 eip 等于一条 jmp 指令，跳转得以实现。

设置方法为调用 wrmsr 指令，这是特权指令。方法如下：把寄存器代号放入 ecx 中，把要设置的内容放在 eds: eax 中。

```
mov ecx, 0x8                    ; Windows XP 下代码段选择子为 8
mov dword ptr eds:[eax], 0x8???????  ; 固定跳转到某个内核地址
wrmsr                           ; 写入
```

以上代码当然是 Windows 系统执行的。然而如果编写了 R0 代码也可以执行，但是在 R3 下不能执行。

值得注意的是，SYSENTER_CS_MSR 总是被设置为 8。Windows 中所有的内核代码段都共用这个描述符——实际计算得到的段基地址总是全 0，这就是为何在 Windows 下虚拟地址总是等于线性地址的原因。

此外，sysenter 执行之后的堆栈段，以及 sysexit 返回后的代码段和堆栈段也都需要确定段选择子。这个在指令手册上已经有规定，所以只需要设置一个 SYSENTER_CS_MSR 就可以了。剩下的在这个选择子之后连续排列，如表 12-2 所示。

表 12-2

段选择子	功 能
SYSENTER_CS_MSR 的值	用于 sysenter 执行后代码段
SYSENTER_CS_MSR 的值+8	用于 sysenter 执行后堆栈段
SYSENTER_CS_MSR 的值+16	用于 sysexit 执行后代码段
SYSENTER_CS_MSR 的值+24	用于 sysexit 执行后堆栈段

当然，从这里可以看出，既然内核代码段的选择子总是 8h，那么堆栈段的选择子只能是 10h，而 R3 代码段的选择子就只能是 1bh 了。

总而言之，sysenter 的过程可以概述如下：

（1）装载 SYSENTER_CS_MSR 到 cs 寄存器。

（2）装载 SYSENTER_EIP_MSR 到 eip 寄存器。

（3）装载 SYSENTER_CS_MSR+8 到 ss 寄存器。

（4）装载 SYSENTER_ESP_MSR 到 esp 寄存器。

（5）切换到 R0 层。

（6）清除 eflags 寄存器中的 VM 标志。

（7）执行 eip 开始的 R0 例程。

而 sysexit 返回的过程除了用到上面表格中的两个段选择子之外，还要加载 edx 到 eip 中，加载 ecx 的值到 esp 中。换句话说，在执行 sysexit 之前，要返回的地址应该保存在 edx 中。概述如下：

（1）SYSENTER_CS_MSR+16 装载到 cs 寄存器。

（2）将 edx 的值送入 eip。

（3）SYSENTER_CS_MSR+24 装载到 ss 寄存器。

（4）将 ecx 的值送入 ESP。

（5）切换回 R3 层。

（6）执行 eip 处的 R3 指令。

上机实践

这里来实际观察一下 Windows XP 从 R0 层切换到 R3 层的方法，以增加感性认识。下面用 WinDbg 来观察实际的 ntdll!NtReadFile 的实现，这是读取文件的 API 函数。毫无疑问，读取文件最终要调用内核 API（一般称为服务例程，本书称为内核 API），换句话说，必须要切换到 R0 级别。请读者打开 WinDbg，在 ndll!NtReadFile 设置断点。当虚拟机中断的时候，来单步跟踪：

```
ntdll!NtReadFile:
001b:7c92e27c    mov     eax,0b7h
001b:7c92e281    mov     edx,offset
    SharedUserData!SystemCallStub (7ffe0300)
001b:7c92e286    call    dword ptr [edx]
001b:7c92e288    ret     24h
001b:7c92e28b    nop
```

很显然，这里是直接调用一个称为 SystemCall 存根的地址，然后你会发现这个地址就是有名的 KiFastSystemCall 调用。所有试图进入 R0 的 API 函数都通过这个调用，而且这个调用仍然是 R3 级的调用。

为了从一个入口就能调用一系列的内核 API，Windows 实际上做了一张跳转表。（在第 13 章“开发 Windows 内核 Hook”中，读者会对此有更多认识。）在调用任何一个内核 API 之前，先把编号填入 eax 中（如上，内核中同名 API，NtReadFile 的编号为 0b7h），之后的内核代码会根据这个编号选择合适的内核的 API 地址进行跳转。

下面来看 KiFastSystemCall 调用，这个函数极短。

```
ntdll!KiFastSystemCall:
001b:7c92eb8b    mov edx,esp
001b:7c92eb8d    sysenter
ntdll!KiFastSystemCallRet:
001b:7c92eb8f    nop
001b:7c92eb90    nop
001b:7c92eb91    nop
001b:7c92eb92    nop
001b:7c92eb93    nop
ntdll!KiFastSystemCallRet:
001b:7c92eb94    ret
```

值得注意的一点是，R3 的堆栈空间如何被传入 R0 代码中，这里用的方法是 mov edx,esp。在内核代码中，直接 mov esp,edx 就可以得到 R3 的堆栈了，完全不用拷贝，

就能直接得到所有的参数。

另一个疑点是返回的地址。sysexit 要正常返回，必须把返回地址放入 edx 中，而这里 edx 却被用来保存了 esp……真是一个令人难以理解的地方。

实际上，R3 的函数如果是正常的函数，当然会把这个函数的返回地址压在栈中。既然栈指针已经得到，那么返回地址也就得到了。只要在执行 sysexit 之前，放入 edx 中即可。

所以单步调试的时候读者会发现，sysenter 执行之后，下一条指令 R3 指令并不是 sysenter 下跟着的一堆 nop，而是 ntdll!NtReadFile 下 call dword ptr [edx]之后的后一条指令，也就是 ret 24h。

以上是 32 位的情况，64 位下有所不同。因为 AMD 首先在 64 位中引入，所以 Intel 也只能跟进，采用和 AMD 同样的指令，使用 syscall 和 sysret 来进行切换。

```
ntdll!NtReadFile:
00000000'77f9fc60        mov     r10,rcx
00000000'77f9fc63        mov     eax,0xbf
00000000'77f9fc68        syscall
00000000'77f9fc6a        ret
```

最后要提的一点是，围绕这个简单的函数，“系统保护软件”和各类外挂、木马进行了各种 Hook 和反 Hook 的种种斗争，这是因为系统默认所有需要进入 R0 的代码都要经过此地。

Windows 中几乎所有 R3 层的 API 最终都通过 KiFastSystemCall 进入 R0 层。

实际上，作为一个软件开发者，自己写 sysenter 来代替这个函数并非不可能；但是想通过 Hook 这个函数就控制进入 R0 的所有入口，显然是不现实的。

下一章读者可以学到 Hook 技术，从而加入到这场混战中来。

第 13 章　开发 Windows 内核 Hook

13.1　XP 下 Hook 系统调用 IoCallDriver

基础知识

Hook（钩子）的意义是将一些系统调用的过程替换成自己编写的函数，这样，系统中任何对该调用的调用都将被拦截到。编程者可以加入自己的处理，来改变或者监控系统的行为。

从这里开始，我希望用新的内核阅读能力做一些更有用的事情。很多安全软件用了系统内核调用 Hook 技术，当然威胁系统安全的软件也一样在使用着它们，这虽然是 MS 所不希望看到的，但却不是我们所可以不关心的。XP 下几乎做任何已经导出的系统调用的 Hook 都非常容易，这正是 Hook 流行的原因。

IoCallDriver 是一个非常重要的调用，在这里可以过滤到所有的系统请求。IoCallDriver 的另一个常用的名字是 IofCallDriver，几乎所有的内核驱动都调用了 IofCallDriver。即使你在开发中写下 IoCallDriver，编译后的代码中也只能发现 IofCallDriver 这个符号。

技术原理

IofCallDriver 的反汇编非常简单。

```
; Exported entry 42. IofCallDriver
; LONG __cdecl IofCallDriver(PDEVICE_OBJECT DeviceObject, PIRP Irp)
public @IofCallDriver@8
@IofCallDriver@8 proc near
jmp ds:_pIofCallDriver
@IofCallDriver@8 endp
```

几乎所有的系统调用都如此：进入之后，立刻出现一个 jmp ...，跳到真实的调用处，这形成了一个系统调用跳转表。这真是一个糟糕的地方，因为只要修改这个地址，一切系统调用的 Hook 都是简单轻松的了。

特殊提示

不过初学者可能会产生一些错误的想法：既然是 jmp ds:_pIofCallDriver，那么我只要把符号_pIofCallDriver 的值修改掉，一切不就 OK 了吗？

任何符号（函数名和变量名）对机器而言都不存在，是 WinDbg 或其他工具根据符号表显示给人类看的，只能看不能改。

但是读者看见的符号，在机器码中并非符号，而是已经编译得出的数字。假设_pIofCallDriver = 8099c000h，那么这条指令本质上是：

```
jmp ds: 8099c000h
```

这个值可以被修改，但是_pIofCallDriver 这个符号无法被修改。假设其他地方也调用了这个符号的话，在这里对值的修改，并不影响其他地方对这个符号的调用，那些地方依然是 8099c000h。

这样一来，如果我们想修改符号的值，似乎唯一的办法是对 8099c000h 进行全面查找替换。但是不幸的是，除了_pIofCallDriver 这个符号之外，其他的数据也可能为这个值。所以结论为：符号的值，不可以修改。

示例代码

下面开始考虑修改 jmp ds: _pIofCallDriver，在 32 位下，jmp 后面直接就是 32 位地址。回顾一下前面看过的反汇编引擎的技术，很容易想到第一个字节是前缀，第二个字节是 jmp 这个操作码，后面 4 个字节就是 32 位的跳转地址。当前要修改它还有一个前提，就是我能找到这个指令本身所在。用 MmGetSystemRoutineAddress 可以得到 IofCallDriver 的地址，而这个 jmp 正是第一条指令。那么写法如下：

```
    // 首先定义一个函数指针类型
    typdef NTSTATUS FASTCALL (*PMY_IOFCALLDIVER_FP)( IN PDEVICE_OBJECT,IN OUT
PIRP);
    // 以下函数用函数 newIofCallDriver 去代替现有的 IofCallDriver
    // 同时返回旧的 IofCallDriver 所跳转的实际地址，如果失败，则返
    // 回空
    PMY_IOFCALLDIVER_F MyHookIofCallDriverXP(
        IN PMY_IOFCALLDIVER_F newIofCallDriver,
        IN BOOLEAN hookOrUnhook)
```

```
{
    UNICODE_STRING functionName;
    PBYTE address;
    static PMY_IOFCALLDIVER_F oldIofCallDriverBody = NULL;
    RtlInitUnicodeString( &functionName, L"IofCallDriver" );

    // 得到 IofCallDriver 的入口地址
    address = MmGetSystemRoutineAddress( &functionName );
    if(address == NULL)
        return NULL;
    if(hookOrUnhook)
    {
        // 获得入口地址后，加 2 字节的地址就是旧的
        // IofCallDriver 的执行体的地址
        oldIofCallDriverBody =
            (PMY_IOFCALLDIVER_F)(*(PLONG)(address + 2));
        InterlockedExchange((PLONG)(address + 2),
            newIofCallDriver);
        return oldIofCallDriverBody;
        }
        else
    {
        // 复原
        if(oldIofCallDriverBody == NULL)
        {
            return NULL;
            InterlockedExchange((PLONG)(address + 2),
                oldIofCallDriverBody);
                return oldIofCallDriverBody;
        }
}
```

以上函数不是线程安全的，应该避免多线程同时调用它，调用后新的 IofCallDriver 将调用你设定的函数，同时你得到了旧的 IofCallDriver 的实现体指针。新的函数调用完后，读者可以选择继续调用旧的或者不再调用而直接结束，从而形成 Hook。

13.2　Vista 下 IofCallDriver 的跟踪

下面是在 Vista 操作系统下对 IofCallDriver 的反汇编结果。阅读解释作为注释写在

右边，以便于读者理解。可以看到，这不是一个简单的跳转过程，而是直接对函数的实现。

```
nt!IofCallDriver:
mov     edi,edi             ;无意义指令
push    ebp                 ;F 指令，备份 ebp
mov     ebp,esp             ;F 指令，备份 esp
push    ecx                 ;F 指令，备份 ecx

; 取一个全局变量 pIofCallDriver 的值放入 eax 中
mov     eax,dword ptr [nt!pIofCallDriver (81931b7c)]
push    esi             ;F 指令，备份 esi
;取第一个参数。第一个参数是一个 PDEVICE_OBJECT device,
;fast call 方式存在 ecx 中。这里相当于 esi = device
mov     esi,ecx

;判断 pIofCallDriver（已存在 eax 中）是否为 0，如果是 0，就跳
;81827e88 地址处；不是 0，则直接调用 pIofCallDriver
xor     ecx,ecx                 ;ecx = 0
cmp     eax,ecx
je      nt!IofCallDriver+0x1d (81827e88)
push    dword ptr [ebp+4]       ;ebp+4 是函数的返回地址
mov     ecx,esi                 ;把参数传回 ecx
call    eax                     ;呼叫 pIofCallDriver
jmp     nt!IofCallDriver+0x63 (81827ecf) ;跳到函数结束处

81827e88:    ;前面判断 pIofCallDriver 为 0 时来这里
dec     byte ptr [edx+23h]    ;实际上是 irp->CurrentLocation--;

;if(irp->CurrentLocation < 0) 就调 bugcheck 报错，否则跳过
;报错的代码
cmp     byte ptr [edx+23h],cl
jg      nt!IofCallDriver+0x30 (81827e9c)

;这一段就是出 BugCheck 报错。程序执行到这里计算机会蓝屏
push    ecx
push    ecx
push    ecx
push    edx
push    35h
call    nt!KeBugCheckEx (818d85ab)
;BugCheck 之后实际就蓝屏了，不可能执行到这里
int     3
```

```
81827e9c:    ;前面不出错时，跳到这里继续执行

; (PBYTE)irp->Tail.Overlay.CurrentStackLocation -= 24
;这个等于 IoSkipCurrentIrpStackLocation 的逆操作
mov      eax,dword ptr [edx+60h]
sub      eax,24h
mov      dword ptr [edx+60h],eax

;这里取得 CurrentStackLocation 的主功能号
mov      cl,byte ptr [eax]
;判断主功能号是否是 IRP_MJ_POWER
cmp      cl,16h
mov      dword ptr [eax+14h],esi
jne      nt!IofCallDriver+0x57 (81827ec3)
mov      al,byte ptr [eax+1]

;如果是 IRP_MN_SET_POWER，就跳 81827eba，去调用
; IopPoHandleIrp
cmp      al,2
je       nt!IofCallDriver+0x4e (81827eba)
; 如果不是 IRP_MN_QUERY_POWER，就跳 81827ec3
cmp      al,3
jne      nt!IofCallDriver+0x57 (81827ec3);
81827eba:    ;
mov      esi,edx          ;似乎是无意义指令，难道用 esi 传参数
call     nt!IopPoHandleIrp (81804fdc)
jmp      nt!IofCallDriver+0x63 (81827ecf)

81827ec3:    ;这里开始是除了电源设置和电源查询之外的处理
mov      eax,dword ptr [esi+8]    ; 取 device->DriverObject.
push     edx                      ;
movzx    ecx,cl
push     esi
call     dword ptr [eax+ecx*4+38h]    ; 调用对应的分发函数
pop      esi
pop      ecx
pop      ebp
ret
nop
nop
nop
nop
nop
```

根据上面的解读，反 C 的结果如下。

```
NTSTATUS FASTCALL IofCallDriver(
    PDEVICE_OBJECT device, PIRP irp)
{
    PIO_STACK_LOCATION irpsp;

    // 首先检查一个全局变量 pIofCallDriver，如果这个不为空
    // 则调这个，否则继续
    if(pIofCallDriver != NULL)
        return pIofCallDriver(device,irp);

    // 移动 irp 的 Current Stack Location.
    irp->CurrentLocation--;
    if(irp->CurrentLocation < 0)
    {
        KeBugCheckEx(0x35,irp,0,0,0,0);
    }
    irpsp = --irp->Tail.Overlay.CurrentStackLocation;

    // 如果是电源设置与查询，则调用特殊的 IopPoHandleIrp
    if(irpsp->MajorFunction == IRP_MJ_POWER)
    {
        if(irp->MiniorFunction ==
            IRP_MN_SET_POWER     ||
            irp->MiniorFunction ==
                IRP_MN_QUERY_POWER)
        {
            return IopPoHandleIrp(device,irp);
        }
    }

    // 否则调用对应的 Device 的 Driver 的分发函数
    return device->DriverObject->DispatchFunctions
        [irpsp->MajorFunction](device,irp);
}
```

上面的反 C 代码供有兴趣的读者研究。这里有个有趣的地方，似乎是微软为了自己使用的方便，留了一个全局变量名字为 pIofCallDriver，只要设置这个指针，IofCallDriver 就会调用这个函数，而忽略其他的处理。一般的跟踪表明这个全局变量都为空。我们可以在这里加上自己的函数的地址来进行处理，这样也可以形成一个 Hook。调用结束后，我们可以设法再跳转回来继续执行。

但是这样的方法过于依赖 IofCallDriver 这个函数的内部实现。pIofCallDriver 这个变量并没有导出，要搜索这个变量就不太容易。而且我将面向未来的 Vista 操作系统，而不是已经确定的 Windows XP。微软随时可能修改这些代码，所以，强烈依赖于具体实现的方法是不可取的。

13.3　Vista 下 inline hook

基础知识

inline hook 方式的基本原理是，动态解析 IofCallDriver 开头的几条指令，并把它们拷贝到另一个地方，同时用一个调用我们的函数的代码加以替换。如果要继续执行旧的 IofCallDriver，在我们的函数调用完毕后，再执行被我们移动过的几条指令，执行完后跳回原来的地方继续执行。具体步骤是：

① 把 IofCallDriver 开头指令拷贝下来，移动到我自己的中继函数里。

② 在 IofCallDriver 开头写入跳转指令跳转到我的中继函数。

③ 中继函数中执行对我自己的 IofCallDriver 钩子函数的调用。

④ 中继函数中执行原来 IofCallDriver 函数中拷贝过来的几条指令。

⑤ 跳转回原来的 IofCallDriver 后开始的跳转点继续执行。

当然要澄清一个误解：并不是只有 Vista 下才能使用 inline hook。在 XP 下，找到真正的 IofCallDriver 的入口之后，进行 inline hook 一样是完全可行的，而且效果比修改跳转地址会更好。

13.3.1　写入跳转指令并拷贝代码

现在要来写代码实现上面的设想。这些都是内核代码，因此读者必须建立一个内核工程。如果忘记了如何用 WDK 编写内核程序，请回头阅读 8.1 节。下面将 Hook 的代码放入 DriverEntry 中，这样驱动一加载，IofCallDriver 就被 Hook。

首先的问题是如何实现写入跳转指令。

跳转用 jmp 就可以实现，但是在计算地址的时候比较麻烦。jmp 后可以指定绝对地址和相对地址，但这段代码我们得写在 IofCallDriver 前部，这样相对地址就变了。此外除了 jmp，很多指令都能实现跳转。有些老手不喜欢用 jmp，因为这容易被其他工具检测到而暴露。

我们必须设法使用 jmp 的绝对地址跳转，或者自己计算相对地址，这都不是很简单。但是我们有反汇编引擎来帮助我们生成机器码（这是前面的反汇编引擎提供的一个相反的功能）。

首先必须查一下 jmp 在指令手册中的详细描述。读者会发现当 jmp 的指令码为 0xea 时，我们可以使用绝对地址，这样就免除了计算相对地址的麻烦。

使用前面的反汇编引擎的汇编功能如下：

```
size_t length;
byte_t code[12];      // 使用一个比较大的空间来容纳未知指令
struct xde_instr myjmp = { 0 };
myjmp.opcode = 0xea; // 填写指令码 eah
myjmp.addrsize = 4;  // 地址长度为 4 个字节（32 位）
myjmp.datasize = 2;  // 选择子作为立即数，长度为 2 个字节
myjmp.addr_l[0] = my_address;     // 填写我要跳转到的地址
myjmp.data_s[0] = 8; // 选择子。内核代码段选择子为 8
length = xde_asm(code,&myjmp);
```

这样一来，一条绝对跳转指令就被写到了 code 变量中。以后可以把 code 中的代码拷贝到 IofCallDriver 的前部。

然后这里的任务仅仅是把这段代码拷贝到 IofCallDriver 的前部，这并不困难，IofCallDriver 的开始地址可以用前面 MmGetSystemRoutineAddress 的方法来获得。之前这些将被覆盖的代码必须先移动到另一个地方，这个地方将位于只读页。回忆前一章对内存保护的介绍，只读页在 R0 模式下也是不允许写的（486 后引入），这会引发系统异常。但是用下面的代码可以清除这一设置。

```
dword_t old_cr0;
_asm
{
    mov eax,cr0
    mov old_cr0,eax
    and eax,0xfffefffff
    mov cr0,eax
}
```

当然最后你还要恢复它。

```
_asm
{
    mov eax,old_cr0
    mov cr0,eax
}
```

这就是 R0 级别的好处。本书已经说过，在 R0 级可以做任何事，这样的限制当然难不住我们。

下面的任务是拷贝代码。前面得到的 jmp 指令为 7 个字节，也就是说，我们至少要拷贝出 7 个字节的代码。我们不能只拷贝 7 个字节，指令长度不定，这可能把一条指令分成两段。于是只好逐条执行进行反汇编，当得到的总的字节数达到或者超过 7 个字节的时候，大功告成。下面假设 IofCallDriver 的开始地址为 start_address。

```
size_t length,total_length = 0;
struct xde_instr code_instr={0};
byte_t *start_address = (byte_t *)MmGetSystemRoutineAddress(…);
while(total_length < 7)
{
    // 反汇编一条指令
    length = xde_disasm(start_address ,&code_instr);
    if(length == 0)          // 如果有指令解析失败，就直接返回失败
        return false;
    total_length += length;  // 计算已经反汇编的指令的总长度
    start_address += length; // 解析地址移动
}
```

得到长度后就简单了，拷贝即可。这些代码将拷贝到所谓的中继函数地址上去。所谓的中继函数将在 13.3.2 节的内容中介绍。中继函数将容纳 IofCallDriver 的开头一些指令，并跳转到我们设置的自己的 IofCallDriver 函数，完毕后继续执行旧的 IofCallDriver 的代码。这中间最好不要调用 C 库函数，同时提高中断级禁止线程切换，以免节外生枝，发生其他的问题。

```
KIRQL irql;
irql = KeRaiseIrqlToDpcLevel(); // 提高中断级
_asm {
    mov esi,start_address       // IofCallDriver 的原来的开始地址
    mov edi,g_new_address       // 这里写入我要拷贝到哪里
    mov ecx,total_length        // 总长度
    cld
    rep movsb                   // 拷贝

    …   // 然后是写入前面的 jmp 指令的过程
```

```
            // 这段代码前面有，故省略
    }
    KeLowerIrql(irql);
    …
```

13.3.2 实现中继函数

下面假设中继函数如下：

```
static __declspec(naked) MyVistaIofCallDriverRelay();
```

这个函数将容纳 IofCallDriver 的开头一些指令，并跳转到已编写的自己的 IofCallDriver 函数，完毕后继续执行旧的 IofCallDriver 的代码。naked 表明这个函数将不生成函数框架指令，这有利于控制生成的代码。

除了耐心细致地做好堆栈平衡工作之外，有人可能疑惑如何容纳旧的部分代码然后执行，实际上你可以这样实现：

```
static __declspec(naked) MyVistaIofCallDriverRelay()
{
    … … // 在这里调用自己编写的新的 IofCallDriver
    _asm {
old_codes:
        // 这里写入足够多的 nop，形成一片空间来容纳拷贝
        // 过来的旧代码
        nop
        nop
        nop
        …
        };
    … …
}
```

不过这样又出来一个新的问题，那就是 old_codes 这个标号对应的具体地址如何传出，以便外面完成拷贝。既然在这个函数内部容易得到这个地址，那么我可以在这个函数执行的早期，把这个地址传入一个叫做 g_new_address 的全局变量里。

C 语言中标号代表一个地址，这个地址可以被写入全局变量中，从而传递到函数外被使用。

```
static byte_t *g_new_address = NULL;
…
```

```
static __declspec(naked) MyVistaIofCallDriverRelay()
{
    _asm push old_codes
    _asm pop g_new_address
    …
        …
        _asm {
old_codes:
            // 这里写入足够多的 nop，形成一片空间来容纳拷贝过
            //来的旧代码
            nop
            nop
            nop
            …
            };
            … …
}
```

这样只要 MyVistaIofCallDriverRelay 执行过一次，就会把 old_codes 这个地址写入全局变量 g_new_address 中。当然，你得在 Hook 之前，执行一次这个函数，并在写 g_new_address 前加一些条件判断，以防止执行后面的代码出错。

含有相对跳转指令的代码块不能被简单地拷贝到其他位置执行，因为拷贝之后跳转的目标地址就不一致了。

这样的 Hook 还是有一些局限的，比如在被拷贝的代码中，不能有各种跳转，否则相对地址会发生错误，这限制了我们拷贝地址的长度，也限制了我们加入其他的处理。比如，我要设置我的函数执行完毕后，是否继续执行后面的代码，或者直接返回。

网上已经有一些库通过反汇编引擎修改各种跳转指令的地址，以便可以拷贝大部分的指令，有需要的读者可以参考。

思考与练习

本章讲述了两种做内核 Hook 的方法，并给出了关键的代码，但是这些代码并不是马上就可以实际运行的代码。而且，本书也不提供光盘或者代码下载的地址。这是因为笔者强烈建议本书的读者自己动手编写代码来熟悉这些过程；否则，阅读本书的意义将失去大半。

下面是留给读者的一个练习：请在网上搜索著名的系统分析工具 IrpMon。这是一

个非常有用的工具，它能显示出系统中的 Irp 的种种信息。该工具是研究 Windows 内核各个驱动之间的关系的有力武器。

但是截至目前的版本，我未发现它支持 Vista。其原因显然是作者只使用了简单的 Hook，没有使用 inline hook，从而在 Vista 下遇到了限制。

留给读者的作业是：开发一个同时支持 XP 和 Vista 版本的 IrpMon。这是一个有趣的挑战，不是吗？

实战篇

实际开发

实战部分是本书最深入和复杂的一部分，包括第 14～17 章。为了让前面练习的成果，在实际应用中产生价值，在这部分我们补充更多的理论知识并尝试用它们去做一点什么。这一部分包括指令分析、硬件基础知识、内核 Hook 的实际开发练习，以及将完成一个用到内核 Hook 的有趣的实例，这个实例有助于计算机阻挡各种病毒和木马的侵袭。

此外，本部分还包括特殊的一章，涉及如何巧妙地编写代码，来防止被其他不受欢迎的读者阅读。这与本书的主旨完全相反，正所谓物极必反。

第 14 章　反病毒、木马实例开发

14.1　反病毒、木马的设想

设计思想

本节讲解的并非是那些日新月异的 rootkit 技术，不是病毒或反病毒技术发展的先锋，而是适合初学者的传统的、古老的，但是有效的安全技术。

一般地认为，病毒通过感染其他的可执行文件而存在；木马则等于是系统中的一个后门，它可能会在使用者不知不觉中与外界取得联系，如泄露机密信息、下载更多非法程序等。当前国内流行的病毒许多集成了木马的技术：会主动从某些网站下载新的病毒代码，或者更多种类的病毒，同时接受远程指令作为“肉鸡”被操作。总而言之，现在已经很难明确地区分病毒和木马了，我们可以以“恶意程序”同等视之。

恶意程序可以做很多的事情，如果使用了 rootkit 技术，则可以做更多的事情。防止被查出、防止被删除，保持“低调”的恶意，然而又带来实际的经济价值（如充当攻击用的僵尸网络或者散发广告）是最好的选择。问题在于，恶意程序的作者未必是态度认真的代码编写者，他们很难保证自己的代码始终保持兼容和“低调”，有些还带着唯恐天下不乱的恶习。这就给用户带来种种困扰：电脑死机蓝屏、性能无节制地损失、文件损坏、无法上网等。

清除一个已经在运行着的恶意程序是极其困难的。如果我有代码运行在内核级，我就可以设法与这个操作系统紧密结合，极难被清除。这就是为何并不是安装了杀毒软件，就能清除掉所有的病毒和木马的原因。查杀和免杀的技术每天都在发展，就像没完没了的猫和老鼠的游戏。

系统被破坏的前提，就是恶意的代码确实被运行了。如果恶意的代码没有被运行，那就不存在破坏性，当然也不可能避免被找到，或者避免被清除。所以实际上，杀毒软件更应该叫做“防毒”软件，防毒的效果远胜杀毒。

Vista 下微软推出了 UAC（User Account Control，用户账户控制）方案，UAC 肯定不是一种完美的安全方案，但它成功地帮助我们免除了很多自动运行的攻击威胁。用户至少需要点一下鼠标，恶意软件才会运行。UAC 的问题是太麻烦，甚至想运行 Vista 自身的程序都会跳出 UAC 来。

现在要探讨的是如何防毒。本文所述的防毒有两个前提：

- **前提 1**：在内核层，我们可以做所有的事。
- **前提 2**：虽然恶意代码运行后会有严重后果，但是恶意代码还没有运行。

这个问题被归结于如何防止恶意代码运行。虽然有人发展了“行为检测”的技术来判断一段代码是否是恶意的，或者可能是恶意的，但是这依然是个高新主题。我只能明确地说：实际上，我无法检测一段代码是恶意的；但是反过来，我知道一些代码不是恶意的。比如，我可以有以下的假定：

- **假定 1**：假定本机并没有感染恶意代码或储存恶意代码，那么，本机上所有的可执行文件都不是恶意的。

正因为我无法判断什么代码是恶意代码，那么这个问题可以改变为：如何做到只允许已知的、非恶意的可执行代码执行。

这是有效的安全技术。如果你明确地知道哪些可执行代码是安全的，那么你就可以只允许执行它们，而阻止执行任何未知的可执行代码。希望下面的假定 2 是成立的。

- **假定 2**：只允许已知的、非恶意的可执行代码执行，并不影响用户正常地使用计算机。

技术原理

下面我们看看实现的原理。

这取决于 Windows 是如何加载一段可执行的代码，并让它进入系统运行的。典型的范例是双击一个.exe 文件，这会导致这个 exe 以及其调用的 dll 文件被加载进入内存执行，虽然还有其他的办法，但这是最经典的一种。要知道相关的知识，可以反汇编 NtCreateProcess。顾名思义，这个内核函数会生成一个进程。

这些文件的格式基本上都是 PE 格式，包括 exe 文件、内核驱动 sys 文件，或者是 dll 文件、ocx 文件，以及其他一些不同软件所带的各种古怪扩展名的插件文件。但是也有一些例外，如古老的 com 文件、DOS 下的 16 位 exe 文件。批处理也是可执行的，但是格式却是文本。

- **假定 3**：一个木马或者病毒必须存在于以上形式的一个文件中或者本身就是以上的一个文件，否则无法运行。（这里除掉了引导区病毒，现在认为古老的引导区病毒已经不再构成威胁。）

另外考虑一下缓冲区溢出病毒。前面已经说过，恶意代码可以仅仅存在于数据文件

中，比如一个 pdf 文件，当相关软件打开这个 pdf 文件的时候，该软件中的缓冲区溢出漏洞导致了这些代码被执行。但是，如果仅仅只能打开这个 pdf 文件的时候才能被执行，这并不构成严重威胁：病毒或者木马必须把自己写成可执行的文件格式或者感染其他可执行的文件，这样才能“经常的”被用户执行，做传播和复制之类的事情。

但是，如果一个病毒永远只感染 pdf 文件，以上假定 3 就不成立了，pdf 文件不是可执行文件。但是实际上应该没有这样的病毒，因为 pdf 阅读器的漏洞往往在发现之后被迅速地修复。如果一个病毒只能感染 pdf 文件，在漏洞被修复之后这个病毒就永远地被终结了。聪明的方法是在此之前把自己写成可执行文件或者感染其他可执行文件，在系统中长期存在下去。

遵循上面的假定，我们这里只研究可执行文件的加载。在 Windows 中，反汇编调试 NtCreateProcess 是最好的办法。跟踪代码并结合自己对系统的理解，对于一个被加载执行的可执行文件，以下的假定是成立的。

- **假定 4：**可执行文件必须被以 FILE_EXECUTE 存取方式打开。
- **假定 5：**可执行映像必须被装载进内存中，这是通过生成一个所谓的 SectionObject 来进行的。

以上假定能否推翻，取决于在系统中能否发现绕过以上假定 4 和假定 5 步骤的漏洞。我不是漏洞专家，故忽略这个问题。可执行文件的打开就是普通的文件的打开方式，这是文件操作。文件操作有很多方法可以捕获，例如在 WDK 的示例中，有 sfilter 的代码，这是一个“文件系统过滤”的驱动程序的框架。你可以用 sfilter 的技术来捕获文件打开的操作，当然也可以使用 NtCreateFile 的 Hook。

以 FILE_EXECUTE 存储方式打开并非一定是为了将可执行文件加载执行，这是一个问题。许多时候打开文件都以完全访问权限打开，这时候就很难判断是在复制一个.exe 文件还是这个文件将被执行了。更重要的是，操作普通的其他文件也会导致这个动作被捕获，所以这种方法很少被人使用。

但是我个人依然是支持这种做法的。检查一个文件是否是可执行文件基本上是可行的，可以依次判断是否是 PE 文件、com 文件、十六进制可执行文件、bat 文件等。如果这个文件并不是可执行文件，那么可以认为是非恶意的，可以直接放行。

此外就是在一个文件被加载的过程中，很可能被连续打开多次，每一次都要做判断是比较烦恼的。对此可以采用第一次判断的结果：如果用户认为是允许的，那么这个可执行文件并非恶意，以后每次都放行它；否则，以后每次都禁止以可执行权限打开它，只允许以读、写、删除权限打开。

这种方法比较复杂，但是很通用，因为即使是在 Vista 64 位版本下，内核 Hook 已

经很困难的情况下，我们依然可以用文件过滤来实现这个检查。比较严重的缺陷是效率，几乎每个文件打开（只要带有可执行权限）时都不得不进行检查。

技术细节

相对精确的方式来自假定 5。如果有人通过 Windows 将一个可执行文件加载执行，一定要生成一个 SectionObject 映射进内存。这个函数简单精确，函数原型如下：

```
NTSTATUS
  ZwCreateSection(
    OUT PHANDLE  SectionHandle,
    IN ACCESS_MASK  DesiredAccess,
    IN POBJECT_ATTRIBUTES  ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER  MaximumSize OPTIONAL,
    IN ULONG  SectionPageProtection,
    IN ULONG  AllocationAttributes,
    IN HANDLE  FileHandle OPTIONAL
    );
```

以上有用的参数是 FileHandle，这是要映射的文件句柄。另一个重要的参数是 SectionPageProtection，如果这个参数被设置为 PAGE_EXECUTE，则是可执行的。DesiredAccess 参数不需要关注。

思考与练习

请读者用以前的知识，来 Hook 这个函数吧。你可以选择普通的 Hook 或者 inline 的 Hook，这也是一个练习。

同时请思考：如何检查一个可执行文件是否是自己允许的合法的可执行文件呢？

14.2 开发内核驱动

14.2.1 在内核中检查可执行文件

技术原理

哈希算法是检查文件完整性的常用方法。我可以简单地对自己允许的所有合法模块

计算其 MD5 值，然后将这些 MD5 值保存起来。当有一个新的模块要被加载时，首先要打开这个可执行文件，计算其 MD5 值。如果结果在自己预先保存的合法 MD5 值之中，那么，这个可执行文件是合法的；否则，这个可执行文件是不合法的，或者是一个本来合法的可执行文件，已经被病毒感染了。

示例代码

下面写代码来实现这个过程。请注意，这次编写的并不是应用程序，而是内核程序。你必须建立一个内核工程，如果忘记了如何用 WDK 编写内核程序，请回头阅读 8.1 节。

现在假设已经 Hook 了 ZwCreateSection，那么现在我们写新的 ZwCreateSection 函数如下：

```
BOOL MyCreateSection (
    OUT PHANDLE SectionHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER MaximumSize OPTIONAL,
    IN ULONG SectionPageProtection,
    IN ULONG AllocationAttributes,
    IN HANDLE FileHandle OPTIONAL)
{
    // passthru 表示“是否放过”。默认是都放过的，只有发现
    // 需要禁止的时候才禁止
    BOOLEAN passthru = TRUE;

    do {
        // 这是设计常识：这个驱动应该可以从外面快速地
        // “禁止”或者“启用”。g_enable 是一个全局变量
        // 在外面可以设定
        if(!g_enable)
            break;
        // 设计常识：可能有一些进程是我们信任的进程，当它
        // 们试图加载可执行文件的时候，不需要检查。一
            // 个典型的例子是这个软件的监控进程
        if(MyCheckProcess(PsGetCurrentProcessId()))
            break;
        // 然后进行正式的检查，检查决定这个可执行文件是否
        // 允许加载
        passthru = MyCreateSectionJudge(
            SectionHandle,
```

```
                DesiredAccess,
                ObjectAttributes,
                MaximumSize,
                SectionPageProtection,
                AllocationAttributes,
                FileHandle);
        } while(0);

        if(!passthru)
        {
            // 如果被禁止了，那么必须返回 DENY
            *SectionHandle = NULL;
            return STATUS_ACCESS_DENIED ;
        }
        // 如果允许的话，直接交给真正的 ZwCreateSection
        return MyOldZwCreateSection(
                SectionHandle,
                DesiredAccess,
                ObjectAttributes,
                MaximumSize,
                SectionPageProtection,
                AllocationAttributes,
                FileHandle);
}
```

上面的 MyCheckProcess 留给读者自己设计，这里可以做得简单也可以做得复杂。用户可以维护一个表：重要的可信任的进程是允许任意加载模块的（假定它们总是没有漏洞，足够的安全），然而一些“漏洞多如蜂窝”的程序则绝不要这样做，比如臭名昭著的 IE 6.0。

IE 很难保证安全性。它是无数的病毒和木马的靶子。这是一个需要重点防范，切不可轻易加入为“信任”的进程。

下面是 MyCreateSectionJudge 的实现。

示例代码

```
BOOLEAN MyCreateSectionJudge (
    OUT PHANDLE SectionHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
```

```
    IN PLARGE_INTEGER MaximumSize OPTIONAL,
    IN ULONG SectionPageProtection,
    IN ULONG AllocationAttributes,
    IN HANDLE FileHandle OPTIONAL)
{
    BOOLEAN passthru= TRUE;
    NTSTATUS status;
    UCHAR file_md5[16] = { 0 };

    // 如果不带有可执行属性，直接退出
    if((SectionPageProtection &
        (PAGE_EXECUTE|PAGE_EXECUTE_READ|
            PAGE_EXECUTE_READWRITE|
            PAGE_EXECUTE_WRITECOPY)
        ) == 0 )
        return TRUE;

    // 然后得到文件的 MD5 值
    status = MyGetMd5Value(FileHandle,file_md5);
    if(!NT_SUCCESS(status))
    {
        // 如果读取 MD5 失败了，为了保险起见，返回禁止
        return FALSE;
    }

    // 检查这个 MD5 是否在合法的 MD5 列表中，如果
    // 在，直接返回 OK
    ret = MyCheckMd5ValueInWhiteList(file_md5);
    if(ret)
        return TRUE;

    // 如果这是一个未知的模块...
    // 见后面的解说.....
    return MyCheckUnknownFile(FileHandle) ;
}
```

编写 MyGetMd5Value()是一个技术问题，请使用 ZwReadFile 读取文件的内容，并使用 MD5 算法计算其 MD5 值。MD5 的算法可以从网上下载到代码，如果在这里提供出来，会占用许多篇幅，这是没有必要的。

ZwReadFile 使用文件的句柄（句柄已经在参数中得到）来读取文件。

下面是软件策略问题：如何编写函数 MyCheckUnknownFile()?

如果对于不认识的可执行文件一律禁止加载，这是可行的，只不过认识的可执行文件必须足够的多，多到足以支撑系统正常运作。但是在测试过程中，这样做显然是不行的，而且给普通用户使用也是不可行的，因为不知道用户会安装什么新软件。一旦用户试图执行新的可执行文件就会被阻止，这是不可接受的。

一个可以选择的方案是：每次有一个可执行模块要加载时，就弹出一个对话框来询问（是不是想起了一些防火墙老是弹出对话框来询问是否允许一个进程访问网络？），这个对话框应该显示调用这个可执行文件的进程名、文件的路径、文件的部分信息，比如版本等。我们可以让用户选择“禁止”，这样这个文件将被禁止加载；也可以选择“信任”，一旦信任，则这个模块的 MD5 将被加入到允许列表中，下次就不需要再次认证了。

当然，弹出一个对话框询问这样的工作绝对不要在内核中做，虽然并非不可能，但是这不符合正确的编程方法，正确的方法是开发一个 Windows 应用程序作为这个软件的“监控”进程。这个应用程序执行着并和内核驱动保持着通信关系，当内核驱动需要用户参与验证的时候，通知监控进程弹出对话框。

14.2.2 在内核中生成设备接口

基础知识

在内核程序中要和应用程序通信的话，有一些其他的工作要做。内核程序必须生成一个“设备对象”，并为这个设备对象生成一个有应用层可以访问的“符号链接”。

设备对象可以使用 IoCreateDevice 生成，符号链接使用 IoCreateSymbolicLink 生成。请在 WDK 的帮助中查找这两个内核 API 的用法和意义。

设备生成之后在应用程序（14.3 节中详细描述的“监控进程”）中可以用 CreateFile 来打开这个设备（访问方法和访问文件相差无几），此外可以通过 ReadFile、WriteFile 和 DeviceIoControl 来进行通信。

为此驱动必须设置自己的分发函数，分发函数就是专门用于处理 Read、Write、DeviceIoControl 等请求的处理函数，分发函数完成这些请求，实现驱动的功能。

示例代码

下面是生成设备并设置分发函数的示例代码。回忆 8.1 节“开始 Windows 内核编程”

中提到的最简单的 C 文件，现在请根据那里的描述建立工程，在 DriverEntry 中加入一些代码：

```
#include "ntifs.h"

// 一个便于定义字符串的宏
#define RTL_STRING_CONST(x)  \
{sizeof(L##x)-2, sizeof(L##x), L##x}

NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg_path)
{
    NTSTATUS status;
    PDEVICE_OBJECT device;
    UNICODE_STRING device_name=
        RTL_STRING_CONST("\\Device\\ MyAntiVirus");
    UNICODE_STRING syb_name=
        RTL_STRING_CONST(L"\\DosDevice\\ MyAntiVirusCDO");

    // 生成设备对象
    status = IoCreateDevice(
        driver,
        0,
        &device_name,
        FILE_DEVICE_UNKNOWN,
        0,
        FALSE,
        &device);

        // 如果不成功，就返回
        if(!NT_SUCCESS(status))
            return status;

        // 生成符号链接
        status = IoCreateSymbolicLink(
            &syb_name,
            &device_name);
        if(!NT_SUCCESS(status))
        {
            IoDeleteDevice(device);
            return status;
        }

    DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = MyCreateClose;
```

```
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
        MyDeviceIoControl;

    // 打开 Hook。这个函数本书示例代码没有实现，请读者根据
    // 前一章“开发 Windows 内核 Hook”自己编写
    MyStartHook();

    // 设备生成之后，打开初始化完成标记
    device->Flags &= ~DO_DEVICE_INITIALIZING;
    return status;
    }
```

上面完成了设备的生成以及分发函数的设置，下面的问题是那 3 个分发函数的编写。应该如何处理打开与关闭请求？在不需要考虑多个进程同步的情况下，只需要直接“允许”任何进程打开和关闭这个设备就可以了。所以处理非常简单：直接成功。

```
NTSTATUS
MyCreateClose(
    IN PDEVICE_OBJECT device,
    IN PIRP           irp)
{
    irp->IoStatus.Information = 0;
    irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

如果读者还不了解 IRP 结构，请查阅 WDK 中的帮助。在驱动开发中，一个 IRP 代表了一个请求，其中包含请求的“功能号”，意味着这是什么类型的请求，还包含了请求的输入、输出缓冲区等更多信息；IoStatus 中包含请求的处理者要填写的返回值。上面函数中的 3 行对 IRP 的处理，是将 IRP 设置为“已成功完成”的方法。

然后是用于处理 DeviceIoControl 的函数 MyDeviceIoControl，这在 14.2.3 节中讲述。

14.2.3 在内核中等待监控进程的响应

基础知识

回到前面的问题。在内核中，本软件“探测”到了一个可执行文件正要被 Windows 加载，此时停下来进行检查，检查的结果是这是一个未知的模块，此时，我们只好把决定权交给用户。监控进程将收到一个通知，弹出一个对话框，让用户决定是否加载。用户点击按钮之后，驱动得到通知，继续执行下去。

在内核中，往往以内核事件进行通知。事件的使用例子如下：

```
// 定义一个事件
KEVENT event;
// 事件初始化
KeInitializeEvent(&event,SynchronizationEvent,TRUE);
……

// 事件初始化之后就可以使用了。在一个函数中，你可以等待某
// 个事件，如果这个事件没有被人设置，那么就会阻塞在这里继续
// 等待
KeWaitForSingleObject(&event,Executive,KernelMode,0,0);
……

// 这是另一个地方，有人设置这个事件。只要一设置这个事件
// 前面等待的地方，将继续执行
KeSetEvent(&event);
```

这个事件不需要重置，每次设置之后，等待函数得以通过，通过的时候自动就重置了，下次等待函数要通过就必须再次设置。有了事件的机制之后，我们就可以做如下的设计了。

当驱动发现一个需要用户来参与判断的请求的时候，就生成一个节点，这个节点中含有要判断的文件的各种信息（也可以只填写该文件的路径，让用户进程去获取信息），此外含有一个事件。这个节点被放入队列中，然后驱动开始等待那个事件。下面本书把这个队列称为“未处理队列”。

用户的监控进程则不断用 DeviceIoControl 来读取节点信息，判断结束之后，再用 DeviceIoControl 来设置一个事件。该事件设置结束了之后，驱动得以继续，并可以删除掉该节点。

示例代码

这需要两种 DeviceIoControl 请求。实际上，一个为读一个为写，也可以用 ReadFile 和 WriteFile 来代替，但是我们还是使用 DeviceIoControl。下面读者请定义两个 DeviceIoControl 的功能号：

```
#define MY_IO_IOCTRL_GET \
    (ULONG)CTL_CODE(FILE_DEVICE_UNKNOWN, \
    0xa01, \
    METHOD_BUFFERED, \
```

```
    FILE_READ_DATA|FILE_WRITE_DATA)
#define MY_IO_IOCTRL_SET \
    (ULONG)CTL_CODE(FILE_DEVICE_UNKNOWN, \
    0xa02,  \
    METHOD_BUFFERED, \
    FILE_READ_DATA|FILE_WRITE_DATA)
```

上面的这个写法有些烦琐，但是这确实是所有 DeviceIoControl 的功能号的定义方法，必须借助 CTL_CODE 宏，其中真正需要我们确定的数字，仅仅是 0xa01 和 0xa02。

应用程序如何调用这些 DeviceIoControl 接口在下一节再描述，这里先介绍在内核驱动中对 GET 的处理。为了编程简便起见，这里总是给应用层返回一个文件全路径，此外还有一个 key（key 的作用在后面介绍）。通信用的缓冲区结构如下：

```
typedef struct MY_GET_PACK_
{
    void *key;                  // key，后面解释
    size_t pathname_size;       //要判断的可执行文件的路径
    wchar_t pathname[1];
} MY_GET_PACK, *PMY_GET_PACK;
```

这个结构就是 DeviceIoControl 的输出缓冲。如果应用程序调用提供的输出缓冲区不够长，则返回失败并返回实际需要的长度。下面是我们的内核驱动中，对 DeviceIoControl 处理的函数。

```
NTSTATUS MyDeviceIoControl(
    PDEVICE_OBJECT   dev,
    PIRP             irp)
{
    // 得到 irpsp 的目的是为了得到功能号、输入/输出缓冲
    // 长度等信息
    PIO_STACK_LOCATION irpsp =
        IoGetCurrentIrpStackLocation(irp);
    // 首先要得到功能号
    ULONG code = irpsp->Parameters.DeviceIoControl.IoControlCode;
    // 得到输入/输出缓冲长度
    ULONG in_len =
        irpsp->Parameters.DeviceIoControl.InputBufferLength;
    ULONG out_len =
        irpsp->Parameters.DeviceIoControl.OutputBufferLength;

    if(code == MY_IO_IOCTRL_GET)
        // 如果是获取请求，让 MyIoCtrlGet 来处理
        return MyIoCtrlGet(irp,out_len);
```

```
    if(code == MY_IO_IOCTRL_SET)
        // 如果是判断结束通知，让 MyIoCtrlSet 来处理
        return MyIoCtrlSet(irp,in_len);

    // 其他的请求我们不接受，直接返回错误。请注意这里返
    // 回错误和前面返回成功的区别
    irp->IoStatus.Information = 0;
    irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

那么现在需要继续编码的是 MyIoCtrlGet 和 MyIoCtrlSet 这两个函数。首先讲解 MyIoCtrlGet，这个函数的任务是：

（1）判断未处理队列中有没有可执行文件判断请求，如果没有，则等待。

（2）如果未处理队列非空，则有请求需要判断。此时判断请求缓冲区长度是否足够，如果不足够，则返回需要的长度。

（3）如果长度足够了，可以让监控进程“得到”一条结果。同时，这个节点被从未处理队列中删除，插入“正在处理的队列”中。

```
NTSTATUS MyIoCtrlGet(PIRP irp,ULONG out_len)
{
    MY_NODE *node;
    ULONG pack_len;
    // 获得输出缓冲区
    PVOID buffer = irp->AssociatedIrp.SystemBuffer;

    // 从队列中取得第一个。如果为空，则等待直到不为空
    while((node = MyGetPendingHead()) == NULL)
    {
        KeWaitForSingleObject(
            &g_my_notify_event,  // 一个用来通知有请求的事件
            Executive,KernelMode,FALSE,0);
    }

    // 有请求了，此时请求是 node。获得 PACK 要多长
    pack_len = MyGetPackLen(node);
    if(out_len < pack_len)
    {
        irp->IoStatus.Information = pack_len;// 这里写需要的长度
        irp->IoStatus.Status = STATUS_INVALID_BUFFER_SIZE;
```

```
        IoCompleteRequest (irp,IO_NO_INCREMENT);
        return irp->IoStatus.Status;
    }

    // 长度足够，填写输出缓冲区
    MyWritePackContent(node,buffer);
    // 头节点从未处理队列移动到正在处理队列
    MyPendingHeadMoveToDealing();
    // 返回成功
    irp->IoStatus.Information = pack_len;// 这里写填写的长度
    irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

不过要注意，以上这个处理完全没有多线程安全性可言。如果有多个线程同时来调用这个 DeviceIoControl，对队列的操作可能崩溃。但是监控进程显然只有一个，只要监控进程是单线程来调用 DeviceIoControl，就没有问题。所以以上的代码对队列并没有加锁。

接下来的任务是，如果用户做了一个决定，必须把这个决定通知驱动。用户的决定不外乎以下几种：总是允许这个模块（加入合法列表）；这次允许（那下次还要判断）；这次禁止（同上）、总是禁止（加入禁止列表）。

此外有必要让用户知道判断的是哪个文件，这就是前面我们的输出中需要一个 key 的原因。这个 key 很简单，可以直接用节点的指针，或者让驱动能找到那个请求节点的任何索引值，这样驱动就可以根据那个 key 知道，哪个请求完成了。

```
// 用户可能的判断结果
enum {
    MY_ACTION_ALWAYS_ALLOW = 1,
    MY_ACTION_ALLOW = 2,
    MY_ACTION_DENY = 3,
    MY_ACTION_ALWAYS_DENY = 4
};

// SET 请求包：通知驱动处理的结果。key 用于驱动把结果和
// 请求对应起来
typedef struct MY_SET_PACK_
{
    void *key;          // 回答的请求的索引
    int action;         // 回答的结果
} MY_SET_PACK, *PMY_SET_PACK;
```

数据结构确定后，下面是 SET 的处理函数。

```
NTSTATUS MyIoCtrlSet(PIRP irp,ULONG in_len)
{
    MY_NODE *node = NULL;
    ULONG pack_len;
    // 获得输入缓冲区
    PVOID buffer = irp->AssociatedIrp.SystemBuffer;
    PMY_SET_PACK my_set_pack = (PMY_SET_PACK)buffer;

    if(in_len < sizeof(MY_SET_PACK))
    {
        // 这里写需要的长度
        irp->IoStatus.Information = sizeof(MY_SET_PACK);
        irp->IoStatus.Status = STATUS_INVALID_BUFFER_SIZE;
        IoCompleteRequest (irp,IO_NO_INCREMENT);
        return irp->IoStatus.Status;
    }

    // 这个函数找到处理队列中相应的节点，设置事件
    // 并从节点中删除它
    MyNotifyAction(my_set_pack->key,my_set_pack->action);

    // 返回成功
    irp->IoStatus.Information = sizeof(MY_SET_PACK);
    irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

当然，读者可能会疑惑，在 ZwCreateSection 的 Hook 处理中，如何处理追加请求这样的事情。本书并不实现队列这样的数据结构，这样的操作请读者利用自己以前的数据结构知识自己完成。回到 14.2.1 节，最后没有完成的函数还有 MyCheckUnknownFile()，现在来考虑这个函数如何写。

```
BOOLEAN MyCheckUnknownFile(HANDLE file)
{
    // 准备一个事件
    KEVENT event;
    // 准备一个 int 型变量，用来接收答复
    int action;
    KeInitializeEvent(&event);

    // 在这里将这个文件的请求加入 Pending 队列。请注意
    // 为了防止多线程，这个追加函数必须用锁同步。请查
    // 阅帮助中的 SpinLock
```

```
    if(!MyAppendFileToPending(FileHandle,&event,&action))
    {
        // 追加失败了！出于安全性考虑，我们返回禁止
        // 这种情况在内存极度枯竭的时候可能发生
        return FALSE;
    }
    // 追加成功后，就等待答复了。首先通知外部，有新请求加入
    // 然后等待回答。下面这句实际也可以在
    // MyAppendFileToPending 内部做
    KeSetEvent(&g_my_notify_event);

    // 等待请求的答复
    KeWaitForSingleObject(
            &event, // 一个用来等待答复的事件
            Executive,KernelMode,FALSE,0);

    // 到这里已经有答复了
    if(action == MY_ACTION_ALWAYS_ALLOW ||
        action == MY_ACTION_ALLOW)
        return TRUE;
    return FALSE;
}
```

思考与练习

上面我们提供了关键的代码，但是还是有一些任务需要读者自己完成：如何完成 MD5 算法？如何保存黑名单、白名单？在硬盘和内存中，用怎样的数据结构，可以快速有效地完成判断？如何实现队列？如何保证队列的多线程安全性？还需要实现的 My 系列的函数还有多个，请读者用自己的编程知识来完成它们，并将这个驱动编译通过，当做本节的练习。

14.3 开发监控进程

基础知识

由于监控进程主要涉及应用程序的开发，所以在本书中占的篇幅并不多，但是这样的一个开发过程并不简单。如果从来没有内核程序和应用程序通信的概念，读者需要理

解一些新的东西。相信读者可以开发一个含有 Windows 对话框的程序，这个程序如果要和一个驱动程序通信，必须打开该驱动程序生成的一个设备。该设备应该有名字，而且拥有一个可以供应用程序打开的“符号链接”。

一个和驱动程序通信的进程应该被开发为一个“服务”程序，因为服务程序拥有管理员权限。要打开驱动生成的设备，一般都需要管理员权限。作为练习可以暂时不这样做，建立一个普通的 MFC 单对话框工程即可，只是执行的时候必须确保以管理员权限执行。

示例代码

读者可以在对话框初始化的过程中打开驱动的控制设备。

```
HANDLE device=CreateFile("\\\\.\\MyAntiVirusCDO",
    GENERIC_READ|GENERIC_WRITE,0,0,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_SYSTEM,0);
if (device == INVALID_HANDLE_VALUE)
{
    // ….打开失败，说明驱动没加载，报错即可
}
```

当然，这是控制设备的符号链接名为 MyAntiVirusCDO 的情况，接下来就可以对这个驱动的句柄调用 DeviceIoControl 了。DeviceIoControl 需要提供的是输入缓冲区、输出缓冲区和功能号。

```
DWORD length = 0;                   // 返回的长度
BOOL ret = DeviceIoControl(device,
    MY_IO_IOCTRL_GET,               // 功能号
    NULL,                           // 这个函数仅仅获取，没有输入
    0,                              // 输入缓冲长度为 0
    out_buffer,                     // 输出缓冲
    out_buffer_len,                 // 输出缓冲的长度
    &length,                        // 返回实际需要的长度
    NULL);
```

这样的调法有可能失败（当缓冲区长度不够的时候），不过此时处理并不困难，根据 length 中返回的长度设置分配一个更长的缓冲区继续尝试即可，这个 DeviceIoControl 可以无限制地调用下去。每次如果有文件需要判断，则返回得到一个 MY_GET_PACK 结构，此时可以弹出对话框让用户去判断；如果没有文件需要判断，则 DeviceIoControl

这个函数不会返回，会阻塞起来直到有请求进入队列为止（请注意上一节内核中的实现）。

一旦用户判断结束，则需要调用 SET 请求来通知驱动。要输入的总是一个 MY_SET_PACK 结构，这个长度倒是固定的，不用担心缓冲区长度问题。代码如下：

```
MY_SET_PACK pack;
DWORD length = 0;                    // 返回的长度

// 填写一个用户处理结果。这里是用户同意确认这个模块
// 总是允许的情况
pack.action = MY_ACTION_ALWAYS_ALLOW;
// 填写在 GET 的时候得到的 key，以便驱动知道这个答复是
// 给谁的
pack.key = key;

BOOL ret = DeviceIoControl(device,
    MY_IO_IOCTRL_SET,                // 功能号
    &pack,                           // 输入缓冲
    sizeof(MY_SET_PACK),             // 输入缓冲长度
    NULL,                            // 输出缓冲。不需要
    0,                               // 输出缓冲的长度为 0
    &length,                         // 返回实际需要的长度
    NULL);
```

思考与练习

本书并没有讲解普通的 Windows 应用程序应该如何开发，比如，如何制作一个对话框等。读者可以用任何开发工具开发 Windows 对话框应用程序（不限于 VC），只要能调用几个主要的 API 函数即可。

本节的练习是完成这个监控进程的开发。

14.4 本软件进一步展望

虽然只是一个示范的用例，但是本软件的设计思想还是有一些借鉴意义的。一般的杀毒软件都是保留病毒的特征码，并在文件打开的时候扫描（一般都是用文件过滤驱动

程序进行实时扫描），这样的特点是只能找到已知的病毒。但病毒花样翻新，不断出现新的、未知的病毒，在加入到病毒特征库之前，这样的病毒是不能被杀毒软件发现的。

本软件体现了另一种思路，那就是并不寻找病毒特征码，而是只保存“合法的可执行模块”，只检查“本模块是否允许执行”，而干脆地禁止一切未知的东西，不管它是不是病毒。

这并不是什么新的思想。实际上，很久以前就有人断言：“安全问题不在于去检查哪些东西是危险的，而在于确认哪些东西是安全的。”

要适合普通用户的软件，这个软件需要有一个专门的机构，这个机构认证世界上所有的“合法软件”，包括它们的不同版本，所含有的所有可执行文件，并把它们的“认证码”打包在“合法码库”中通过自动更新下载。这个过程酷似杀毒软件不断下载最新的病毒码库，这个工作庞大而复杂。实际上，已经有很多信息安全厂商在“认证”这世界上所有的恶意程序，但到底是恶意程序多还是合法软件多，这个问题我无法回答。

实际上这样的认证机构是有的，Windows 的很多组件都有证书签名，被合法机构的证书签名过的组件被认为是安全的。有一些应用程序也有。Windows 有一套检查签名认证的机制，不过，应用到所有的软件开发商身上，那规模就太大了，现在还不合时宜。

但是在一个企业内部应用，情况就不同了。因为一个企业的员工所使用的软件往往是有限的，而且原则上必须是通过公司核准的。

因此一个公司可以把公司内部需要使用的软件的所有可执行文件的“合法码”都计算出来，然后放在服务器上下载。用户则可以使用规定范围内的软件，而且几乎永远不会感染病毒。

即使用户自己从网上下载软件进行安装，也将被禁止，不管里面是否含有病毒，甚至 IE 浏览网页时“不经意”地安装的各类插件、各种利用漏洞的目录也会被自动禁止掉。所有网站都注定成为“安全”的软件，只是可能有一些控件不能被安装，所以一些功能无法使用。

如果用户需要安装新的软件，则需要报告公司批准，得到批准之后，相关的“合法码”也被加入到库中，自动更新之后，这个软件就可以安装了。

实际上，这样的安全软件已经存在，并应用于一些国内外的企业中。

第 15 章　Rootkit 与 HIPS

本章现在开始简要介绍 Rootkit 和 HIPS（主机入侵检测）。这些领域内聚集着 Windows 最底层的黑客、最孜孜不倦的内核研究者以及最邪恶的病毒制造者，此外或许还有安全软件以及 Windows 内核的作者们。

Rootkit 的字面意义是“根权限工具”。根权限是指在计算机系统上的软件最高权限，拥有根权限就可以直接操作或者修改操作系统的最底层功能。这是一类技术的称呼。在 Windows 上，不考虑具体用途的情况下，一个可控制的内核程序就可以称为一个 Rootkit。但是在这一章范围内，Rootkit 特指使用了 Rootkit 技术的病毒，这类病毒和传统病毒的不同在于：它往往通过加载一个驱动或者其他的手段，使部分代码或者全部代码都在内核中运行。

在这种情况下，简单地扫描病毒是没有意义的。杀毒软件需要检测内核中是否有非法代码在运行，这样的工作一般交给 HIPS（主机入侵检测）来完成。在 Windows 上，显然 HIPS 本身也必须运行在内核层，否则不可能检测 Rootkit 的存在。

和前一章举出的例子不同。Rootkit 必须要被系统执行之后，才能起到它的作用。如果一个 Rootkit 还没有被系统加载执行，那么它的一切高深的功能都是空谈。而 HIPS 则完全必须考虑：在 Rootkit 被加载执行之后，如何检测到这个入侵并及时阻止，让这个入侵失去作用。

换句话说，双方都已经被系统加载执行，现在是正面进行 PK 的时候了。这就产生了一个难题：双方都是在 Windows 内核中运行的程序，那么，互相 PK 的结果是什么呢？答案是没有结果。

举一个简单的例子。一个 Rootkit 拥有一个功能：能生成一个隐藏的进程。这个功能非常有用处，因为许多病毒或者木马甚至外挂程序都不希望自己的进程被杀毒软件、反作弊软件或者其他安全软件看到。至于它的技术原理，取决于安全软件是如何看到进程的，比如调用了 PsSetCreateProcessNotifyRoutine（这个函数在后面的内容中有详细的解释），它能设置一个回调函数来监控进程的产生。

如果对 PsSetCreateProcessNotifyRoutine 除了参数和返回值外一无所知，显然病毒的作者是无法知道如何绕过这个函数所设置的监控的。但是，如读者现在所具备的技术水平，他们可以通过反汇编这个函数来取得未公开的信息，然后将设置好的回调函数屏蔽掉。

到此时，作为 HIPS 的作者，显然就不能继续用这样简单的方式来查看所有的进程了。但是更多的手段可以被找出来：系统中有没有什么全局的结构中保存了所有的进

程？直接从里面读出来不是更直接？或者有没有什么更底层的没有公开过的函数指针或者结构内容能被替换掉？除非 Rootkit 的作者也知道这些，否则……由于这种原因，Rootkit 和 HIPS 的作者都不会公开在软件中实际用到的最新的技术。发表在网上的一般都是过时的或者不是非常重要的“点拨”。但是它们之间的 PK 无限持续下去，永远也没有结果。

15.1 Rootkit 为何很重要

本节不会提出任何实用的开发 Rootkit 的技术，只举出最广为人知的例子，来让读者了解 Rootkit 技术的存在和它的工作方式，同时让读者了解 Rootkit 研究的重要性。

可以假定，一个 Rootkit 刚刚问世的时候，是没有传统的杀毒软件能认出它的（之所以说是传统的，是由于一些新型杀毒软件已经加入了识别未知 Rootkit 病毒的能力，这是一个新的发展趋势）。那么如何保护自己，使之即使在今后也不被识别出来，就是一个重要的工作。

无论 Rootkit 以何种方式保存，它一定要设法保留在硬盘上，否则系统重新启动，那些代码就不复存在了。传统的病毒写一个文件或者感染其他的文件，这都容易被查出：不明的文件或者现有的文件被修改了，扫描那些文件就可能发现问题。既然 Rootkit 有根权限，那么它可以做更多的事情，比如将文件隐藏。

隐藏文件有很多种方法。但是如果要加一个文件过滤驱动的话，那动作就太大了，一般的都是通过修改内核某些数据结构，或者对内核调用做 Hook 来进行。对文件系统数据结构深入了解的作者，可能在文件系统中某些隐蔽的部位（比如某个 NTFS 的流中）隐藏大量的数据。

一个程序可能需要创建进程或者线程来使自己始终得到执行。在系统中出现不明的进程和线程是一个危险的信号，杀毒软件可能检测它们的来源，计算机的使用者也可能警觉。和隐藏文件具有同样的方式，Rootkit 可以向系统输出伪装过的进程列表，绕过一些进程和线程的监控手段（本节后面有例子）。

可以想象如下的一个 Rootkit：

（1）**稳定的执行**。使“用户”从来感觉不到系统不稳定或者效率异常低下。

（2）**安静的传播**。从一台机器到另一台机器，没有网络拥堵，没有异常的攻击包到处发送，但是悄悄地遍布了大量的计算机中。

（3）**无踪可循**。看不到异常的文件，看不到异常的进程或者线程，看不到系统某个组件被改写，杀毒软件不知道它的存在。

（4）**远程可控**。虽然平时不动声色，但是一旦接收到远程的指令（比如一个来自网络的特殊的网络包），就开始执行破坏计划，破坏操作系统，毁灭硬盘数据、销毁自己等。

研究 Rootkit 以及对抗 Rootkit 的 HIPS，是关系到国家安全的大事。

这是有力的战争武器。一旦战争爆发，也许一个国家的大多数民用计算机就可以在几小时内因为接收到攻击指令而全部瘫痪。普通的 PC 修复也许还是比较简单的，但是各种大型的服务器、各种机要部门的关键电脑、保存有重要数据的备份计算机，一旦被破坏，修复是极其困难的，而且有些数据几乎就是不可能恢复的。

计算机病毒和它们的作者们

早在 1998 年就开始使用计算机的读者肯定对 CIH 病毒以及它的作者陈盈豪记忆犹新。

陈盈豪的确做了一些开创性的工作：CIH 似乎是计算机历史上首个可以破坏硬件的病毒（不知道会不会也是最后一个），此外，Windows 98 的“貌似安全”的体系也在瞬间坍塌无疑。大家都知道了微软搞出的应用-内核安全体系只是个摆设，因为 CIH 确实是在应用层启动后轻易就得到了内核权限。

一般认为程序只能破坏其他的程序，但是人们又把主板上用来保存一些最基础代码和信息的那一小块 BIOS 当做硬件来对待。因为它完蛋了主板也就完蛋了，只能拆下来送到厂商那里去重刷。这块东西在 CIH 之前的时代，很多 PC 上都是可写的。

当然陈盈豪没有客气，直接写入了垃圾数据，使许多人抱着无法启动的电脑去找厂商的麻烦。BIOS 的损坏并非每个维修点的人都明白如何修理。从此技术发展了，现在大多数机器上的 BIOS 大都是有写保护的，对于 Windows 也不再那么容易获得内核权限了。

武汉男生的熊猫烧香则是一个彻底的传统型的应用层病毒。作者甚至都不会使用 C 语言。它没有获得内核权限。它关闭杀毒软件的方法非常简单：寻找所有带有各种杀毒软件字眼的窗口全部发送关闭消息。感染局域网内其他计算机的方法也是简

单地通过网络邻居写入的，仅仅是因为大家都常用的密码它都尝试了一遍！此外它写入 U 盘。Windows 有一个极其失败的“自动播放”特性能自动执行 U 盘上写在一个配置文件中的 EXE 程序。当然这个被指定为病毒，这个病毒运行极其不稳定：消耗资源，系统因为非法操作而重启，网络变得缓慢。不知道算不算简单有效的“山寨病毒”？

15.2 Rootkit 如何逃过检测

技术原理

前面提到存在一个内核 API 函数，可以监控进程的诞生。之所以提到这一点，是因为有一个时期，杀毒软件的内核部分常常以这两个函数为基础，进行安全性的检查。比如一个进程诞生的时候，检查是否是已知的病毒进程；加载 DLL 的时候，检查里面有没有病毒码。前面提到的函数是 PsSetCreateProcessNotifyRoutine，这个函数的原型如下：

```
NTSTATUS
PsSetCreateProcessNotifyRoutine(
    IN PCREATE_PROCESS_NOTIFY_ROUTINE  NotifyRoutine,
    IN BOOLEAN  Remove
);
```

其中 NotifyRoutine 是一个回调函数。假设设定 Remove 为 FALSE，则执行这个函数的结果是，把 NotifyRoutine 注册成了一个回调函数。这个回调函数的原型如下：

```
VOID
(*PCREATE_PROCESS_NOTIFY_ROUTINE) (
    IN HANDLE  ParentId,
    IN HANDLE  ProcessId,
    IN BOOLEAN  Create
);
```

这个函数被回调的时候，ParentId 是父进程 Id，而 ProcessId 则是系统要通知我们的进程 Id。这个进程被创建或者被注销的时候，这个回调函数被调用。

Create 表示是创建还是注销。TRUE 表示创建。

有些读者会疑惑，只得到进程的 Id，如何检查这个进程（比如 EXE 文件路径如何

获得？命令行参数如何获得？）呢？在 WDK 之前，这些都可以通过一些非公开的技术手段来获得。但是在 WDK 之后，如果使用 PsSetCreateProcessNotifyRoutineEx，则可以比较容易地得到这些参数。

PsSetCreateProcessNotifyRoutineEx 的不同之处在于，回调函数的参数发生了改变：

```
VOID
CreateProcessNotifyEx (
    __inout PEPROCESS  Process,
    __in HANDLE  ProcessId,
    __in_opt PPS_CREATE_NOTIFY_INFO  CreateInfo
);
```

当 CreateInfor 不为空的时候，这些信息被提供在 CreateInfo 中。

PS_CREATE_NOTIFY_INFO 的结构如下：

```
typedef struct _PS_CREATE_NOTIFY_INFO {
  __in SIZE_T  Size;
  union {
     __in ULONG  Flags;
     struct {
        __in ULONG  FileOpenNameAvailable : 1;
        __in ULONG  Reserved : 31;
     };
  };
  __in HANDLE  ParentProcessId;
  __in CLIENT_ID  CreatingThreadId;
  __inout struct _FILE_OBJECT  *FileObject;
  __in PCUNICODE_STRING  ImageFileName;
  __in_opt PCUNICODE_STRING  CommandLine;
  __out NTSTATUS  CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

既然如此，作为杀毒软件的开发者，就可以在进程加载的时候，检查 ImageFileName，或者直接得到 FileObject 来扫描其中是否含有病毒特征码了。

此外进程加载的每个 DLL 也需要检查。这由另外一个函数提供，原型如下：

```
NTSTATUS
    PsSetLoadImageNotifyRoutine(
        IN PLOAD_IMAGE_NOTIFY_ROUTINE  NotifyRoutine
    );
```

这个函数也追加一个回调函数。当有 DLL 模块被进程加载的时候，这个回调函数

会被系统调用。回调函数 NotifyRoutine 的原型如下：

```
VOID
(*PLOAD_IMAGE_NOTIFY_ROUTINE) (
    IN PUNICODE_STRING  FullImageName,
    IN HANDLE  ProcessId,
    IN PIMAGE_INFO  ImageInfo
);
```

这样 DLL 加载的时候，杀毒软件的开发者就可以从 FullImageName 中得到 DLL 文件的名字，可以预先对这个文件进行扫描。

代码分析

一个 Rootkit 可能会打算防止杀毒软件用以上的方法扫描一些隐秘的进程或者模块文件。那么 Rootkit 的作者为了达到这个目的会怎么做呢？Hook 上面两个函数是一个可行的途径（请读者回忆前面学习过的开发内核 Hook 的章节）。只是直接 Hook 内核调用是太明火执仗了，很多杀毒软件配备的 HIPS 都会关注被 Hook 的内核调用。那么不要 Hook，先看看 PsSetCreateProcessNotifyRoutine 的内部实现是比较好的入手。下面是用 WinDbg 跟踪到的 Vista 32 位版本中 PsSetCreateProcessNotifyRoutine 的实现过程。这是检验读者所学的好机会。请调试代码的同时，自己分析一下它做了什么？

Rootkit 的技术通过调试 Windows 内核和杀毒软件而来，而杀毒软件的 HIPS 技术通过调试 Windows 内核和调试 Rootkit 而来。

```
kd> u PsSetCreateProcessNotifyRoutine
nt!PsSetCreateProcessNotifyRoutine:
81a1c9f3 8bff          mov     edi,edi
81a1c9f5 55            push    ebp
81a1c9f6 8bec          mov     ebp,esp
81a1c9f8 5d            pop     ebp
81a1c9f9 eb0a            jmp
nt!PspSetCreateProcessNotifyRoutine (81a1ca05)
81a1c9fb cc            int     3
81a1c9fc cc            int     3
81a1c9fd cc            int     3
```

很明显，这个 PsSetCreateProcessNotifyRoutine 不是要领，因为直接跳转到 PspSetCreateProcessNotifyRoutine 去了，此外什么都没干。那么再看看 PspSetCreateProcessNotifyRoutine 的跟踪：

```
nt!PspSetCreateProcessNotifyRoutine:
81a1ca05 8bff            mov     edi,edi
81a1ca07 55              push    ebp
81a1ca08 8bec            mov     ebp,esp
81a1ca0a 51              push    ecx

;这里有必要研究一下[ebp+0ch]到底是第几个参数？参数是在
;PsSetCreateProcessNotifyRoutine 被调用之前，倒序压入
;堆栈的。回忆一下函数原型
;
;NTSTATUS
;PsSetCreateProcessNotifyRoutine(
;    IN PCREATE_PROCESS_NOTIFY_ROUTINE  NotifyRoutine,
;    IN BOOLEAN  Remove);
;
;先压入的是 Remove，然后压入 NotifyRoutine（每个参数都是占
;4 个字节的），最后压入返回地址。然后跳转到
;PsSetCreateProcessNotifyRoutine。此时，Remove 实际上是
;esp+8h。进入 PsSetCreateProcessNotifyRoutine 之后连续有
;push 和 pop，但是刚好平衡。esp 没有变动。再跳转
;PspSetCreateProcessNotifyRoutine。此时出现了 push ebp，
;那么 esp 等于减少了 4。之后 mov ebp,esp，也就是说 ebp = esp
;之后的不用管了。Remove 就应该是[ebp+8h+4h]，也就是[ebp+0ch]
;了。顺便可以得到结论是，[ebp+08h]是第一个参数，也就是
;NotifyRoutine。所以下面这句是比较 Remove 是否为 FALSE
81a1ca0b 807d0c00        cmp     byte ptr [ebp+0Ch],0
81a1ca0f 53              push    ebx
81a1ca10 56              push    esi
81a1ca11 57              push    edi

;如果 Remove 为 FALSE 则跳转，否则继续。那么从这里可以看到，下面
;应该是处理“移除”回调函数，而 81a1ca88 则处理了“安装”回调函数
81a1ca12 7474            je
nt!PspSetCreateProcessNotifyRoutine+0x83 (81a1ca88)

;这里开始“移除”回调函数
;首先清 0 ebx
81a1ca14 33db            xor     ebx,ebx
;edi = PspCreateProcessNotifyRoutine。后面会看到，其实
; PspCreateProcessNotifyRoutine 是一个数组
81a1ca16 bf80919081      mov
edi,offset nt!PspCreateProcessNotifyRoutine (81909180)

;压入参数后调用 ExReferenceCallBackBlock。等于是
```

```
; ExReferenceCallBackBlock(PspCreateProcessNotifyRoutine)
;从后面看还可以知道这里是一个循环的开始点
81a1ca1b 57              push    edi
81a1ca1c e863a10600      call
nt!ExReferenceCallBackBlock (81a86b84)

;很明显，检查返回值。如果为 NULL，则跳 81a1ca4d
81a1ca21 8bf0            mov     esi,eax
81a1ca23 85f6            test    esi,esi
81a1ca25 7426            je
nt!PspSetCreateProcessNotifyRoutine+0x48 (81a1ca4d)

;这是 ExReferenceCallBackBlock 返回了非 NULL 的情况
;下面拿参数 NotifyRoutine 来进行比较。请注意 esi 是
; ExReferenceCallBackBlock 的返回值。虽然这个函数没有
; 公开的文档，这时大家也应该猜出，它的返回值是一个指针
;而且这个地址之后 4 个字节开始保存的就是一个回调函数的地址
;下面拿要移除的回调函数地址和旧有的回调函数地址比较
81a1ca27 8b4d08          mov     ecx,dword ptr [ebp+8]
81a1ca2a 394e04          cmp     dword ptr [esi+4],ecx

81a1ca2d 8b4608          mov     eax,dword ptr [esi+8]

;如果比较不等，则跳 81a1ca44
81a1ca30 7512            jne
nt!PspSetCreateProcessNotifyRoutine+0x3f (81a1ca44)

;这是比较之后相等的处理。此时看 ExReferenceCallBackBlock
;返回地址之后第 8 字节开始是否为 0
81a1ca32 85c0            test    eax,eax
;如果不是 0，则跳到 81a1ca44
81a1ca34 750e            jne
nt!PspSetCreateProcessNotifyRoutine+0x3f (81a1ca44)

;否则，调用 ExCompareExchangeCallBack，读者应该是可以
;猜出的，这个意思就是说，把 NULL 写到这个位置里去了
81a1ca36 56              push    esi
81a1ca37 33c9            xor     ecx,ecx
81a1ca39 8bc7            mov     eax,edi
81a1ca3b e867a00600       call
nt!ExCompareExchangeCallBack (81a86aa7)

;这里检查返回值。如果不是 0，则跳到 81a1ca5d。实际上 81a1ca5d
;后面的代码意味着成功结束了
```

```
81a1ca40 84c0          test    al,al
81a1ca42 7519          jne
nt!PspSetCreateProcessNotifyRoutine+0x58 (81a1ca5d)

;如果返回了 0，一般地，status 为 0 表示成功，应该要结束了
;但是看完后面的代码发现不是。ExCompareExchangeCallBack
;返回非 0 才表示成功
;esi 保存着原来 ExReferenceCallBackBlock 返回的地址。一般
;地，有 ExReferenceCallBackBlock 就有
; ExDereferenceCallBackBlock 来收尾
81a1ca44 8bce          mov     ecx,esi
81a1ca46 8bc7          mov     eax,edi
81a1ca48 e86aa20600     call
nt!ExDereferenceCallBackBlock (81a86cb7)

;ExDereferenceCallBackBlock 调用完之后没有检查返回值
;但是这里 ebx 加上 1，然后 edi 加上 4，让 ebx 和 0ch 比较，这些
;都很像一个循环语句
81a1ca4d 43            inc     ebx
81a1ca4e 83c704         add     edi,4
81a1ca51 83fb0c         cmp     ebx,0Ch
;注意跳转回 81a1ca1b 了。此时 edi 加了 4，等于指针后移 4 个字节
;换句话说，从 81a1ca1b 到这里是明显的一个循环
81a1ca54 72c5          jb
nt!PspSetCreateProcessNotifyRoutine+0x16 (81a1ca1b)

;这里开始是一个错误处理，正常不会到这里来。0C000007Ah
;就是 STATUS_PROCEDURE_NOT_FOUND。这就是说循环结束了，但是
;还没有找到一个能插入回调函数的位置。跳 81a1caca，实际上就是直
;接返回
81a1ca56 b87a0000c0     mov     eax,0C000007Ah
81a1ca5b eb6d          jmp
nt!PspSetCreateProcessNotifyRoutine+0xc5 (81a1caca)

;请注意这句之前是 jmp，充分说明了这一句和前面是不连续的。实际上，这是从
;前面检查 ExCompareExchangeCallBack 的返回值不为 0 的时候，跳转过来
;的。这也就是设置成功，然后比较马上就结束了。下面做了一些事情，请读者自
;己翻译
81a1ca5d b8b0919081     mov     eax,
offset nt!PspCreateProcessNotifyRoutineCount (819091b0)
81a1ca62 83c9ff        or      ecx,0FFFFFFFFh
81a1ca65 f00fc108      lock xadd dword ptr [eax],ecx
81a1ca69 8d049d80919081 lea     eax,
nt!PspCreateProcessNotifyRoutine (81909180)[ebx*4]
```

```
81a1ca70 8bce          mov     ecx,esi
81a1ca72 e840a20600     call
nt!ExDereferenceCallBackBlock (81a86cb7)
81a1ca77 8bce          mov     ecx,esi
81a1ca79 e802a00600     call    nt!ExWaitForCallBacks (81a86a80)
81a1ca7e 56            push    esi
81a1ca7f e8df9f0600     call    nt!PspExitApcRundown (81a86a63)

;成功返回的时候，总是跳转到这里。这里首先清 0 eax,也就是设置
;返回值为 STATUS_SUCCESS,然后跳转到返回的地方 81a1caca
81a1ca84 33c0          xor     eax,eax
81a1ca86 eb42          jmp
nt!PspSetCreateProcessNotifyRoutine+0xc5 (81a1caca)

; 下面处理安装。安装也是一个循环过程。和上面类似，这里不注释了，请读者
; 自己理解
81a1ca88 33ff          xor     edi,edi
81a1ca8a 57            push    edi
81a1ca8b ff7508         push    dword ptr [ebp+8]
81a1ca8e e89a9f0600     call    nt!ExAllocateCallBack (81a86a2d)
81a1ca93 8bd8          mov     ebx,eax
81a1ca95 3bdf          cmp     ebx,edi
81a1ca97 7507          jne
nt!PspSetCreateProcessNotifyRoutine+0x9b (81a1caa0)
81a1ca99 b89a0000c0     mov     eax,0C000009Ah
81a1ca9e eb2a          jmp
nt!PspSetCreateProcessNotifyRoutine+0xc5 (81a1caca)
81a1caa0 be80919081     mov     esi,
offset nt!PspCreateProcessNotifyRoutine (81909180)
81a1caa5 6a00          push    0
81a1caa7 8bcb          mov     ecx,ebx
81a1caa9 8bc6          mov     eax,esi
;注意，ExCompareExchangeCallBack 返回成功的时候不为 0，跳 81a1cad2
81a1caab e8f79f0600     call    nt!ExCompareExchangeCallBack (81a86aa7)
81a1cab0 84c0          test    al,al
81a1cab2 751e          jne
nt!PspSetCreateProcessNotifyRoutine+0xcd (81a1cad2)
;这里显然是循环
81a1cab4 83c704         add     edi,4
81a1cab7 83c604         add     esi,4
81a1caba 83ff30         cmp     edi,30h
81a1cabd 72e6          jb
nt!PspSetCreateProcessNotifyRoutine+0xa0 (81a1caa5)
81a1cabf 53            push    ebx
```

```
81a1cac0 e89e9f0600      call    nt!PspExitApcRundown (81a86a63)
81a1cac5 b80d0000c0      mov     eax,0C000000Dh

;很多处理都是填好 eax（返回值）之后跳转到这里来的。这里也就是直接返回了
81a1caca 5f              pop     edi
81a1cacb 5e              pop     esi
81a1cacc 5b              pop     ebx
81a1cacd 59              pop     ecx
81a1cace 5d              pop     ebp
81a1cacf c20800          ret     8

;这里是安装成功跳转过来的。非常简单，将 PspCreateProcessNotifyRoutineCount
;锁定加 1
81a1cad2 33c9            xor     ecx,ecx
81a1cad4 b8b0919081      mov     eax,
offset nt!PspCreateProcessNotifyRoutineCount (819091b0)
81a1cad9 41              inc     ecx
81a1cada f00fc108        lock xadd dword ptr [eax],ecx
81a1cade a1f8909081      mov     eax,
dword ptr [nt!PspNotifyEnableMask (819090f8)]

; 然后，判断 PspNotifyEnableMask 的第 2 位是否为 0
81a1cae3 a802            test    al,2
; 如果第 2 位不为 0，则跳转到 81a1ca84，直接返回成功了
81a1cae5 759d            jne
nt!PspSetCreateProcessNotifyRoutine+0x7f (81a1ca84)

;否则，把 PspNotifyEnableMask 第 2 位置 1 再返回成功
81a1cae7 b8f8909081      mov     eax,
offset nt!PspNotifyEnableMask (819090f8)
81a1caec f00fba2801      lock bts dword ptr [eax],1
81a1caf1 eb91            jmp
nt!PspSetCreateProcessNotifyRoutine+0x7f (81a1ca84)
81a1caf3 90              nop
81a1caf4 cc              int     3
81a1caf5 cc              int     3
81a1caf6 cc              int     3
81a1caf7 cc              int     3
81a1caf8 cc              int     3
81a1caf9 cc              int     3
```

上面研究了这个函数的构成，这时读者会发现原来自己也可以干一些事情。比如，把别人注册的回调函数去掉（既然可以注册，就可以 Remove）。Remove 有一个必要条件，就是必须知道要 Remove 的回调函数的指针。换句话说，如果知道了别人注册过的

回调函数指针，就可以去掉它。对于一个 Rootkit，如果 Remove 掉了这些回调函数，杀毒软件可能就会悄无声息地被禁用掉，而且可能不会弹出任何错误消息。

示例代码

这里涉及的一个问题是如何获得系统中已经注册的进程监控回调函数。实际上，上面的反汇编已经给出了答案。如果把前面的 Remove 一个回调函数的关键部分用 C 语言写出来，读者会看得更清楚。

```
PUCHAR pt_unknown;          // 由于 ExReferenceCallBackBlock 的返回值
                            // 是一个不明类型指针，所以用 PUCHAR

// PspCreateProcessNotifyRoutine 显然是一个全局数组指针。数组
// 大小为 0xC，里面保存有所有的已经设置的回调函数。而且类型也是不明
PUCHAR pt_loop = (PUCHAR)PspCreateProcessNotifyRoutine;
……
if(Remove)
{
    // 如果 Remove 为 TRUE(非 0)
    for(i=0;i<0xC;i++)
    {
        pt_unknown =
            ExReferenceCallBackBlock(pt_loop);
        if(pt_unkown != NULL)
        {
            if((void *)(pt_unknown + 4) ==  (void *)NotifyRoutine)
            {
                // 如果 pt_unknown + 4 等于 NotifyRoutine，则
                ……
            }
            ExDereferenceCallBackBlock(pt_unknown);
        } // if end
        pt_loop += 4;
    } // for end
} // if(Remove) end
……
```

那么显然清楚要得到那些指针，就是要得到 PspCreateProcessNotifyRoutine。然后再 ExReferenceCallBackBlock 一下，将返回值移动 4 个字节就得到了。这个过程真是简单。但是遇到的第一个困难是，PspCreateProcessNotifyRoutine 没有导出，如何找到这个地址呢？

要找到未知的地址，总是要从已知的地址入手。目前唯一已知的地址是 PsSetCreateProcessNotifyRoutine，这个地址是可以用 MmGetSystemRoutineAddress 获得的。实际上得到的地址应该是 81a1c9f3（对比前面的反汇编代码）。nt!PspCreateProcessNotifyRoutine 的地址在后面出现了，是 81909180，这个作为 mov 指令的第二个参数出现。那么，第一种方法就是硬编码。硬编码是怎么求的？实际上是扳手指数出来的。请数出从 81a1c9f3 开始，经过了多少个字节开始出现了 PspCreateProcessNotifyRoutine 的地址，把这个硬编码记住。

这种办法不好的地方在于在不同版本的 Windows 系统上都要数一遍，然后根据不同版本的 Windows 进行版本判断再填入具体的数据。这似乎非常毛糙。但是，令人吃惊的是，这往往是 Rootkit 和 HIPS 常用的做法。如果学习一下著名杀毒软件卡巴斯基的 HIPS 驱动，往往就发现大堆的硬编码。

其次就是 ExReferenceCallBackBlock 之类的函数的地址了。求得方法同上。

更优秀一点的办法也许是反汇编引擎对 PsSetCreateProcessNotifyRoutine 这个函数开始的地址进行反汇编。然后大致地理解代码，认为第几个 call 之后的函数地址一定是 ExReferenceCallBackBlock，第几个 mov 之后的第二个参数一定是 PspCreateProcessNotifyRoutine。这样通用性稍微好一点，但是也不能完全摆脱版本相关性。MS 随时可能修改这些代码，也许一个补丁包下来这些就变了。那就干脆别想那么多了，多写一些硬编码吧。

把别人设置的回调函数直接设置为 NULL 并不是最好的方案。这样的话所有的回调都不起作用了，被察觉的可能性大大增加。更优秀的方案或许是 inline hook 那些回调函数。在里面增加一段处理：如果碰到想逃过检查的进程，则装作没看见，还是正常进行检查，这样最大程度地避免了被察觉。这里不举出 inline hook 的例子，因为前面已经用大量的内容详细地介绍过内核 Hook。

以上绝对不是什么优秀的 Rootkit 手段，只是为了说明 Rootkit 之类的技术是如何被研究出来，而举出的一个用于说明研究手段的例子。用上面的方法去绕过杀毒软件是不可行的，因为这样的实现手段太老了，而且当代的杀毒软件也不可能仅仅通过增加 PspCreateProcessNotifyRoutine 来进行进程监控。

15.3 HIPS 如何检测 Rootkit

HIPS 要检测 Rootkit，有以下几个方面：

（1）检查进程和线程。

（2）检查文件。

（3）检查内核是否被改动。

（4）检查自己的代码是否被改动。

（5）检查注册表。

要检查进程和线程，显然不能简单地用前面的 PsSetCreateProcessNotifyRoutine 的方式。这是微软公开的方法，虽然简单好用，但是显然是各路 Rootkit 要绕过的基本要求。换句话说，如果是用来对付 Rootkit 的，等于什么都没有做。

作者在这里无法拿出完美的解决方案，实际上也没有人可以拿出来。Rootkit 和 HIPS 在互相斗争之中推动技术的发展，但是总体上是从更上层到更下层。在 Windows 的源代码中，从最上层请求生成一个进程，到最下层的实现（实际上到了最下层，就已经看不到进程了，只知道做了乱七八糟的许多操作），中间不知道经过多少个层次，涉及多少代码。如果要监控进程的生成与消失，当然是要在这一个树状调用中，寻找必经的一环，或者没有遗漏的多个点，设置 Hook，或者直接读取其数据的变化。而 Rootkit 如果要绕过这个手段，则应该在已经被监控的层次之下进行，自己来实现部分本来可以直接调用的功能。如此互相比拼的结果，使技术越来越底层，Rootkit 和 HIPS 的实现也越来越脱离微软公开的接口，深入到那些从来没有公开过的底层实现之中。

Rootkit 和 HIPS 谁能取胜？这基本取决于谁做得更底层。

网上一篇流传很广的文章，叫做《AVP 主动防御之隐藏进程》。AVP 指的是著名的杀毒软件卡巴斯基。卡巴斯基的进程监控能力是很强的，不容易轻易绕过。这篇文章介绍的是卡巴斯基监控进程的方法。资料很老了，我不知道新版的卡巴斯基是如何做的，但是估计已经有新的方法出来了。

《AVP 主动防御之隐藏进程》的出处是：

http://blog.vckbase.com/windowssky/archive/2008/02/18/30866.html#32686

这种老方法是 Hook 了 KiSwapContext 这个调用，KiSwapContext 是被 KiSwapThread 调用的。实际上，KiSwapThread 这个调用到底起什么作用，从名字看来应该是和线程有关的。不过读者会发现线程的生成一定会调用 KiSwapThread。下面是作者调试时捕获的一个调用栈。

```
kd> kb 100
ChildEBP RetAddr  Args to Child
f9c3ed0c 804fa73e ff676980 00000000 ffffffff nt!KiSwapThread
f9c3ed44 8060a200 00000003 f9c3eda0 00000001
nt!KeWaitForMultipleObjects+0x284
f9c3edac 805c5a06 00000000 00000000 00000000
nt!ExpWorkerThreadBalanceManager+0x5e
f9c3eddc 80541fa2 8060a1a2 00000000 00000000
nt!PspSystemThreadStartup+0x34
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16
```

调用栈是这样的：最下面是最先调用的函数，从这里看也就是 KiThreadStartup。然后 KiThreadStartup 中调用了 PspSystemThreadStartup，接着调用了 KeWaitForMultipleObjects。最后 KeWaitForMultipleObjects 中调用了 KiSwapThread。实际上，如果 Hook 了 KiThreadStartup，显然也可以起到捕获线程生成的作用。

但是此时，Rootkit 的作者就可以反汇编 KiThreadStartup，然后自己来实现它，这样对 KiThreadStartup 的监控就不起作用了。如果要对抗这种手段，只好监控更下层的东西。假设 Rootkit 绕过 KiThreadStartup 生成线程，那么一定要自己实现 KiThreadStartup 中的内容。全部自己实现太麻烦了，多半还是要调用下面的一些调用的。所以 KiSwapThread 或者 KiSwapContext 就更隐蔽了。调用树是非常复杂的，每个可能的层次上都可能被监控。除非专门针对 AVP 进行研究，否则很难有把握知道应该如何绕过它。

不过即使 Hook 了 KiSwapContext，也未必彻底解决问题。马上有人拿出了自己写代码实现线程调度的方案，有兴趣的读者可以看如下的网址：

http://hi-tech.nsys.by/33/

除了监控进程之外，还有对含有病毒文件的扫描与删除的问题。由于种种隐藏文件的方法存在，因此直接扫描磁盘，自己解析文件系统来读取文件列表是比较彻底的方法，但是非常的麻烦。无论如何，只要使用系统提供的调用来检查文件的存在，就有可能被 Rookkit 修改，从而将文件隐藏起来。

然后是注册表的检查。注册表的访问也是同理。有人开发了直接读取文件数据来解析注册表内容的模块，避免调用注册表相关的系统调用，比如 ZwCreateKey（这些都有可能被 Rootkit 修改其返回结果）。但是微软从来未公开注册表的文件格式，也没有保证过不修改这些格式，因此这些方法或多或少都有一些不稳定性存在。

最后是检查内核是否被改动。检查整个内核是否被改动是不大可能的。内核太大了，而且改动内核也并非总是非法的行为。为此，HIPS 只好关注一些重点位置，比如内核 API 有没有被 Hook？这些 API 开始地址的几个指令有没有被修改过？当然，更重要的是保护自己，比如自己的调用有没有被 inline hook？自己设置的回调函数有没有被替换掉？

如果要详细写 Rootkit 和 HIPS 的技术，不但本书的篇幅无法承受，而且作者本人的技术水平也远远不够。但是相信读者已经掌握了自己研究的基本方法，有兴趣的读者请自己努力探索吧。

专门介绍 Rootkit 中文书籍我只见过一本。有兴趣的读者可以去寻找这本书：

中文名：ROOTKITS —— Windows 内核的安全防护

作者：Butler,J / Hog lund,G

出版社：清华大学出版社

第 16 章　手写指令保护代码

16.1 混淆字符串

基础知识

物极必反。我们已经研究了阅读 Windows 内核的方法，现在开始讨论在我们自己的驱动编码中采用特殊的编码方法，来简单地防止反汇编阅读。这是有趣的一种事态：一方面我们研究如何阅读别人的（尤其是 MS 的）代码；另一方面，我们不得不采取措施保护自己的技术不被他人简单地窃取。

我这里要用到的这种方法不同于加壳或者加密。加壳和加密的方法比较单一，从而容易被人以同样单一的方法整体解密，同时内核驱动中进行加密稍显复杂，难以做得稳定，而且我手头没有成熟的整体加密的代码。实际上代码如果在运行，则一定必须被解密，因此从内存中得到解密的代码并不困难。所以我并不太倾向于使用加壳或者加密的方案。

刘涛涛的“扭曲变换”是根本的防止逆向解决方案，但是需要付出巨大的努力才得以实现，而且他也没有公开代码供大家研究，而是采取了收费服务的方法。不过这也是一个好消息，这使我们依然可以轻松地阅读许多国内、国外人制作的内核程序的汇编代码，并从中获得知识。

下面的一些方法是比较简单的。这样的做法显然不能完全防止逆向，但是能起到一定的“遮掩”作用。好处是随时可以做，而且可以个人根据代码的重要程度，决定哪些部位做，哪些部位不做。无论有多简陋，有胜于无。

刘涛涛的扭曲变换

由于破解技术的发展，实际上大部分商业软件的合法性验证机制都变成了仅具有装饰性，而且一些难以开发的底层软件的技术细节，往往在发行不久就被人理解并发表在网上。这成为许多开发者厌恶的事（当然另一些开发者则乐于此道）。

于是如何让编译后的机器代码更难以理解就成为一个切实有用的研究方向。编译器有效地优化了代码，但这仅仅是将代码变得更简单。一般地说，简单的东西总是比复杂的要容易理解，虽然和源代码的关系变得模糊了。

国内著名技术牛人刘涛涛在网上发表“扭曲变换”的思想。这个思想和“优化”完全相反，主旨不是让代码更简单、效率更高，而是让代码更加复杂——毫无疑问效率也更低。但是相比技术秘密的泄露，效率的损失其实并不那么重要了。

这种复杂性有着无限的潜力。比如，读者计划一次从北京到上海的旅行，最近的直线航行路线当然只有一条。但是如果想要路线更长更复杂的话，则完全有无限种可能：你可以先从北京抵达纽约，在纽约休息后前往南极，然后北上到达撒哈拉沙漠，最后再前往上海。复杂的代码使破解者无法找到验证相关代码的所在，也使试图窃取技术的读者一无所获。

刘涛涛的工具可以将编译后的 obj 文件进行扭曲，链接后生成的可执行文件变得极难阅读，而且可以反复地执行下去：一段代码可以无限地变大变长——功能却是和原来等价的。这真是超级 BT 的技术。

技术细节

在代码中，出现了直接字符串是非常不妥的一件事。往往这些常数字符串在反汇编的时候会直接被人看见，对反工程者是最好的引导。

以下的代码都会暴露我们的字符串：

```
char *str = "mystr";
wchar_t buffer[2] = { L"hello,world."};
UNICODE_STRING my_str;
RtlInitUnicodeString(&my_str,L"hello,world");
```

隐藏这些常数字符串并不能完全防止逆向工程，但是毫无疑问的是，会给逆向工程增加很多麻烦。

隐藏这些字符串的手法是：在写代码的时候并不写字符串的明文，而是书写密文，总是在使用之前解密。这样的手法的后果是，在静态反汇编的时候，反工程者是看不见字符串的。但是，反汇编者显然会在调试的时候看见字符串。

在调试的时候才看见合法的字符串比静态反汇编的时候看见字符串要麻烦许多，因为一般只有对一段代码有兴趣才去调试它。而之所以对那段代码有兴趣，有些时候是因为看见了感兴趣的字符串。全部跟踪调试所有的代码，是艰巨的任务，只有具有重大价值的目标才值得那样去做。

想保护代码，请首先保密常数字符串。

不过有点要注意的是，你不能把所有的字符串都用同样的一招进行加密，至少不能用相同的密钥；否则，也许一个简单的解密程序就把你所有的字符串恢复为明文了。

考虑：

```
char *str = "mystr";
```

改为：

```
char str[] = { 'm' ^ 0x12, 'y' ^ 0x12, 's' ^ 0x12 ,'t' ^ 0x12,'r' ^ 0x12,'\0'
^ 0x12}
```

如果认为异或是最简单的加密方法，那么 0x12 就是这里的密钥。现在编译出来的字符串已经面目全非了，当然，要解密这个字符串需要一个函数。

```
char *dazeEstr(char *str, char key)
{
    char *p = str;
    while( ((*p) ^= key) != '\0')
    {   p++;   }
    return str;
}
```

你可以试试静态反汇编下面的代码：

```
char *str = { 'm' ^ 0x12, 'y' ^ 0x12, 's' ^ 0x12 ,'t' ^ 0x12,'r' ^ 0x12,'\0'
^ 0x12};
dazeEstr(str,0x12);
printf(str);
```

当然你通过分析一定会知道 printf 的结果。调试也可以知道结论，但是比直接用眼睛可以看见可是麻烦多了。当然这样写代码也有些让人抓狂，但是，你总是可以先按自己的本来的习惯写完代码，然后把关键的字符串这样处理。

下面是一个宽字符的处理方法。

```
wchar_t *dazeEstrW(wchar_t *str, wchar_t key)
{
    wchar_t *p = str;
    while( ((*p) ^= key) != '\0')
    {   p++;   }
    return str;
}
```

下面的代码：

```
UNICODE_STRING my_str;
RtlInitUnicodeString(&my_str,L"hello,world");
```

其实总是可以改为：

```
wchar_t buffer[] = { L"hello,world."};
UNICODE_STRING my_str;
RtlInitUnicodeString(&my_str,buffer);
```

那么加密的写法是：

```
wchar_t buffer[] = {
   L'h' ^ 0x3a,L'e' ^ 0x3a,L'l' ^ 0x3a,L'l' ^ 0x3a,
   L'o' ^ 0x3a,L' ^ 0x3a,'L'w' ^ 0x3a,L'o' ^ 0x3a,
   L'r' ^ 0x3a,L'l' ^ 0x3a,L'd' ^ 0x3a,L'.' ^ 0x3a,
   L'\0' ^ 0x3a};
UNICODE_STRING my_str;
RtlInitUnicodeString(&my_str,dazeEstrW(buffer,0x3a));
```

比较明显的缺陷是书写常数字符串的时候变得麻烦。我一直在追寻更简单的写法，但是遗憾的是，我还没有找到。怎么说呢？如果你觉得隐蔽是值得的，那就这样做。你甚至可以用更复杂的加密算法，只要你能算出密文，然后填写在代码常数中，不过那样修改代码变得太困难了。如果你能写一个预编译工具自动修改代码，确实是一个好办法，不过对于一种仅仅防止肉眼直观看到字符串的方式，更复杂的加密方法往往没必要，因为无论多复杂的算法，解密算法都很容易在你自己的代码里找到。

示例代码

下面我们用一个实例来试试。

写一个简单的驱动如下：

```
#include <ntifs.h>

NTSTATUS DriverEntry(
   PDRIVER_OBJECT driver,
   PUNICODE_STRING reg)
{
   UNICODE_STRING my_str;
   RtlInitUnicodeString(&my_str,L"hello,world");

   DbgPrint("%wZ",&my_str);
```

```
    return STATUS_SUCCESS;
}
```

以上函数的 Release 版本用 IDA 反汇编的效果是这样的：

```
MyDriverEntry  proc near
DestinationString= UNICODE_STRING ptr -8
    push    ebp
    mov     ebp, esp
    push    ecx
    push    ecx
    push    offset SourceString    ; "hello,world"
    lea     eax, [ebp+DestinationString]
    push    eax                    ; DestinationString
    call    ds:RtlInitUnicodeString
    lea     eax, [ebp+DestinationString]
    push    eax
    push    offset Format          ; "%wZ"
    call    DbgPrint

    pop     ecx
    pop     ecx
    xor     eax, eax
    leave
    retn    8
MyDriverEntry  endp
```

上面的情况，很容易读懂这段代码打印了一个"Hello，world"。现在把 C 代码改成下面这样的：

```
wchar_t *dazeEstrW(wchar_t *str, wchar_t key)
{
    wchar_t *p = str;
    while( ((*p) ^= key) != '\0')
    {   p++;   }
    return str;
}

NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    wchar_t buffer[] = {
        L'h' ^ 0x3a,L'e' ^ 0x3a,L'l' ^ 0x3a,
        L'l' ^ 0x3a,L'o' ^ 0x3a,L'r' ^ 0x3a,
```

```
        L'w' ^ 0x3a,L'o' ^ 0x3a,L'r' ^ 0x3a,
        L'l' ^ 0x3a,    L'd' ^ 0x3a,L'.' ^ 0x3a,
        L'\0' ^ 0x3a};
    UNICODE_STRING my_str;
    dazeEstrW(buffer,0x3a);
    RtlInitUnicodeString(&my_str,buffer);
    DbgPrint("%wZ",&my_str);
    return STATUS_SUCCESS;
}
```

IDA 反汇编的结果变成下面这样：

```
MyDriverEntry  proc near
DestinationString= UNICODE_STRING ptr -28h
SourceString    = word ptr -20h
var_1C          = word ptr -1Ch
var_1A          = word ptr -1Ah
var_18          = word ptr -18h
var_16          = word ptr -16h
var_14          = word ptr -14h
var_12          = word ptr -12h
var_10          = word ptr -10h
var_E           = word ptr -0Eh
var_C           = word ptr -0Ch
var_A           = word ptr -0Ah
var_8           = word ptr -8
var_4           = dword ptr -4

    push    ebp
    mov     ebp, esp
    sub     esp, 28h
    mov     eax, dword_13000
    mov     [ebp+var_4], eax
    push    3Ah
    lea     eax, [ebp+SourceString]
    push    eax
    mov     [ebp+SourceString], 52h
    mov     word ptr [ebp-1Eh], 5Fh
    mov     [ebp+var_1C], 56h
    mov     [ebp+var_1A], 56h
    mov     [ebp+var_18], 55h
    mov     [ebp+var_16], 48h
    mov     [ebp+var_14], 4Dh
    mov     [ebp+var_12], 55h
    mov     [ebp+var_10], 48h
```

```
    mov     [ebp+var_E], 56h
    mov     [ebp+var_C], 5Eh
    mov     [ebp+var_A], 14h
    mov     [ebp+var_8], 3Ah
    call    sub_11000

    lea     eax, [ebp+SourceString]
    push    eax             ; SourceString
    lea     eax, [ebp+DestinationString]
    push    eax             ; DestinationString
    call    ds:RtlInitUnicodeString
    lea     eax, [ebp+DestinationString]
    push    eax
    push    offset Format  ; "%wZ"
    call    DbgPrint

    pop     ecx
    pop     ecx
    mov     ecx, [ebp+var_4]
    xor     eax, eax
    call    sub_110CD

    leave
    retn    8
MyDriverEntry  endp
```

这样一来，就多少有些挑战性了。调试的时候，当然可以看见正确的字符串，但是静态反汇编的时候，要看明白那段 52h、5fh、56h、55h…是什么就不是一件简单的事情了。遗憾的是，这里用了 RtlInitUnicodeString 和 DbgPrint。我们的读者还是很容易了解这段代码做了什么。下面尝试隐藏那些函数存在的信息。

16.2 隐藏内核函数

基础知识

对于那些对技术有兴趣而进行发掘的破解者来说（出于兴趣而不是处于巨大的经济利益），很多时候，调用的 API 函数是除了字符串之外，对他们最重要的引导之一。一些有经验的 Windows 内核程序反汇编者，对程序流程的兴趣不大，但是看看几个关键

的调用，就已经明白了十之八九。所以，让对方在静态反汇编的时候看不到直观的 Windows 内核 API 函数调用，是非常有意义的。

代码分析

对于 RtlInitUnicodeString 这样简单的函数，本质上没有必要隐藏。但是要隐藏也比较简单，虽然我的方法比较蠢：自己写一个，但由于这个函数涉及字符串的混淆处理，后面会发现写一个是很有意义的。

```
VOID dazeRtlInitUnicodeString(
    IN OUT PUNICODE_STRING  DestinationString,
    IN PWCHAR SourceString,
    IN ULONG length,
    IN WCHAR key,
    )
{
    if(key != 0)
        dazeEstrW(SourceString,key);
    DestinationString->Buffer = SourceString;
    DestinationString->Length = length - sizeof(WCHAR);
    DestinationString->MaximumLength = length;
}
```

上面这个函数如果要当普通的 RltInitUnicodeString 用，那么 key 填写为 0，用法如下：

```
WCHAR buf[] = { L"Hello,world" };
UNICODE_STRING str;
dazeRtlInitUnicodeString(&str,buf,sizeof(buf),0);
```

这样反汇编的时候当然看不见 RtlInitUnicodeString 这个调用了，这是显然的。

需要隐藏的一般都是关键调用，这些调用给人以对这个系统体系的结构理解的指导。比如说如下一个驱动：

```
NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    UNICODE_STRING name_str;
    RtlInitUnicodeString(
        &name_str, L"\\FileSystem\\Filters\\MyFilterCDO" );
    PDEVICE_OBJECT my_cdo;
```

```
    status = IoCreateDevice(
                            driver,
                            0,
                            &name_string,
                            FILE_DEVICE_UNKNOWN,
                            FILE_DEVICE_SECURE_OPEN,
                            FALSE,
                            &my_cdo );
    return status;
}
```

这个驱动生成了一个有名字的控制对象，这个信息反汇编一下一定会马上真相大白。下面我们 DIY 那个关键的调用——其实我们没有 DIY，只是封装了一下，并且用 MmGetSystemRoutine 获得函数地址。

动态获得内核 API 的地址可以避免内核 API 在编译出的 sys 文件的导入表中出现。保密的程度取决于动态获取的方法。

```
typedef NTSTATUS
  (*DAZE_IoCreateDevice)(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,
    IN BOOLEAN  Exclusive,
    OUT PDEVICE_OBJECT  *DeviceObject
    );

NTSTATUS
  dazeIoCreateDevice(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,
    IN BOOLEAN  Exclusive,
    OUT PDEVICE_OBJECT  *DeviceObject)
{
    wchar_t name_buf[] = {
        'I' ^ 0x98,
        'o' ^ 0x98,
        'C' ^ 0x98,
```

```
        'r' ^ 0x98,
        'e' ^ 0x98,
        'a' ^ 0x98,
        't' ^ 0x98,
        'e' ^ 0x98,
        'D' ^ 0x98,
        'e' ^ 0x98,
        'v' ^ 0x98,
        'i' ^ 0x98,
        'c' ^ 0x98,
        'e' ^ 0x98,
        '\0' ^ 0x98};
    UNICODE_STRING name_str;
    DAZE_IoCreateDevice function_pt;
    dazeRtlInitUnicodeString(
       &name_str,name_buf,
       sizeof(name_buf),0x98);
    function_pt =
        (DAZE_IoCreateDevice)MmGetSystemRoutineAddress(
          &name_str);
    if(function_pt != NULL)
       return function_pt(
          DriverObject,DeviceExtensionSize,
          DeviceName,DeviceType,
          DeviceCharacteristics,
          Exclusive,DeviceObject);
    return STATUS_UNSUCCESSFUL;
}
```

反汇编的结果是，驱动里的 Import 表中 IoCreateDevice 这个名字消失了。由于用了前面的字符串处理方法，字符串中也看不到这个信息。以上这个函数反汇编的结果是这样的：

```
sub_11054     proc near
SystemRoutineName= UNICODE_STRING ptr -2Ch
var_24  = dword ptr -24h
var_20  = word ptr -20h
var_1E  = word ptr -1Eh
var_1C  = word ptr -1Ch
var_1A  = word ptr -1Ah
var_18  = word ptr -18h
var_16  = word ptr -16h
var_14  = word ptr -14h
var_12  = word ptr -12h
```

```
var_10  = word ptr -10h
var_E   = word ptr -0Eh
var_C   = word ptr -0Ch
var_A   = word ptr -0Ah
var_8   = word ptr -8
var_4   = dword ptr -4
arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch
arg_8   = dword ptr  10h
arg_C   = dword ptr  14h
arg_10  = dword ptr  18h
arg_14  = dword ptr  1Ch
arg_18  = dword ptr  20h

    push    ebp
    mov     ebp, esp
    sub     esp, 2Ch
    mov     eax, dword_13000
    push    98h
    mov     [ebp+var_4], eax
    push    1Eh
    lea     eax, [ebp+var_24]
    push    eax
    lea     eax, [ebp+SystemRoutineName]
    push    eax
    mov     word ptr [ebp+var_24], 0D1h
    mov     word ptr [ebp+var_24+2], 0F7h
    mov     [ebp+var_20], 0DBh
    mov     [ebp+var_1E], 0EAh
    mov     [ebp+var_1C], 0FDh
    mov     [ebp+var_1A], 0F9h
    mov     [ebp+var_18], 0ECh
    mov     [ebp+var_16], 0FDh
    mov     [ebp+var_14], 0DCh
    mov     [ebp+var_12], 0FDh
    mov     [ebp+var_10], 0EEh
    mov     [ebp+var_E], 0F1h
    mov     [ebp+var_C], 0FBh
    mov     [ebp+var_A], 0FDh
    mov     [ebp+var_8], 98h
    call    sub_11024

    lea     eax, [ebp+SystemRoutineName]
    push    eax             ; SystemRoutineName
```

```
    call    ds:MmGetSystemRoutineAddress
    test    eax, eax
    jz      short loc_110F7

    push    [ebp+arg_18]
    push    [ebp+arg_14]
    push    [ebp+arg_10]
    push    [ebp+arg_C]
    push    [ebp+arg_8]
    push    [ebp+arg_4]
    push    [ebp+arg_0]
    call    eax
    jmp     short loc_110FC

loc_110F7:
    mov     eax, 0C0000001h
loc_110FC:
    mov     ecx, [ebp+var_4]
        call    sub_11176
        leave
        retn    1Ch
    sub_11054     endp
```

示例代码

以上的反汇编结果可以确定调用了某个函数，但是究竟是什么函数，需要进行解密或者动态调试，直接阅读的困难度就大大增加了。

写一个封装的隐藏型函数还是有一定工作量的，但是通过使用宏定义，可以让工作简化，而且好处是，这些工作不会浪费。如果你写了一个头文件和一个 C 文件，那么只要加入到你的任何工程中，那个工程就可以享受“隐藏函数”的服务。这些工作是值得的。

以下这个宏定义把剩余工作变成简单流程。

```
#define DAZE_FUNC_BODY(func_name,key) \
    UNICODE_STRING name_str; \
    DAZE_##func_name function_pt; \
    dazeRtlInitUnicodeString( \
        &name_str,name_buf,sizeof(name_buf),key); \
    function_pt = \
        (DAZE_##func_name) \
```

```
            MmGetSystemRoutineAddress(&name_str); \
    if(function_pt != NULL) \
        return function_pt(
```

把函数原型从帮助中拷贝出来，稍加修改（加两个括弧、一个星号和一个 DAZE_ 前缀）如下：

```
typedef NTSTATUS
     (*DAZE_IoCreateDevice)(
   IN PDRIVER_OBJECT  DriverObject,
   IN ULONG  DeviceExtensionSize,
   IN PUNICODE_STRING  DeviceName  OPTIONAL,
   IN DEVICE_TYPE  DeviceType,
   IN ULONG  DeviceCharacteristics,
   IN BOOLEAN  Exclusive,
   OUT PDEVICE_OBJECT  *DeviceObject
   );
```

然后开始封装，封装固定的方法如下：

```
NTSTATUS
  dazeIoCreateDevice(
   IN PDRIVER_OBJECT  DriverObject,
   IN ULONG  DeviceExtensionSize,
   IN PUNICODE_STRING  DeviceName  OPTIONAL,
   IN DEVICE_TYPE  DeviceType,
   IN ULONG  DeviceCharacteristics,
   IN BOOLEAN  Exclusive,
   OUT PDEVICE_OBJECT  *DeviceObject)
{
   wchar_t name_buf[] = {
      'I' ^ 0x98,
      'o' ^ 0x98,
      'C' ^ 0x98,
      'r' ^ 0x98,
      'e' ^ 0x98,
      'a' ^ 0x98,
      't' ^ 0x98,
      'e' ^ 0x98,
      'D' ^ 0x98,
      'e' ^ 0x98,
      'v' ^ 0x98,
      'i' ^ 0x98,
      'c' ^ 0x98,
      'e' ^ 0x98,
```

```
        '\0' ^ 0x98);
    DAZE_FUNC_BODY(IoCreateDevice,0x98)
          DriverObject,DeviceExtensionSize,
          DeviceName,DeviceType,
          DeviceCharacteristics,
          Exclusive,DeviceObject);
    return STATUS_UNSUCCESSFUL;
}
```

首先写加密字符串来表示函数名，然后用宏，接着填上所有的参数，最后返回不成功即可。

如果你调用的这个函数不想轻易让人看见，就这样做吧。

上面的手法只“遮盖”了字符串和函数调用。下面再增加一些简单方法来“混淆”流程与数据处理等。

16.3 混淆流程与数据操作

16.3.1 混淆函数出口

基础知识

程序的主体流程取决于函数之间的互相调用。一般来说，一个独立的函数内部，流程不可能太复杂（很少有人愿意把一个函数写得太长，那样不容易维护）。那么，破坏流程的可阅读性，就应该首先混淆函数的调用框架。下面主要使用各种无害花指令来混淆函数的调用框架。

花指令往往是指一些没有实际意义的指令，执行它们对最后的结果并不会导致错误，但是阅读或者反汇编分析的时候，会带来不小的困难，甚至干扰 IDA 这样的反汇编工具。读者会很快发现，干扰 IDA 远比想象的容易。前面已经学过，类似 ret 这样的指令，是从堆栈中弹出一个地址并跳到该地址。那么，如下的两句实际与 jmp 等同（这个在前面 NX 保护漏洞的描述中已经有描述）：

```
push MyCode1
ret
```

和 jmp MyCode1 不同的是这里含有一个 ret，所以 IDA 很容易把这个地方当做一个函数结尾。所以，有些人用这样的指令来“伪装”成函数结尾，导致 IDA 分不清真正的函数结尾的位置。

当然这相当于一个 jmp。jmp 的好处见以下的一段代码：

```
jmp MyCode1
…    ; 这里你可以插入任何人能读懂或者不能读懂的东西
MyCode1:
…    ; 这里你插入之前正常的代码
```

上一段代码是无害花指令。执行到 jmp 的时候，后面的语句已经没有任何意义了。你可以插入任何东西，比如无数的 call\jmp\nop\int 3，或者其他任何稀奇古怪的数据都没有关系，反正它们是不会被执行的。但是，它们一定会被 IDA 反汇编，而且阅读者也很有可能需要一段时间才明白，试图去读这一段是没有意义的。

示例代码

还有一些无害花指令是更简单的，比如 push eax, pop eax 之类。当然你可以把它们结合起来做得更令人眼花缭乱一点，比如下面的一个宏：

```
// 伪装的函数结束地址
#define DZ_CAN_RUN_1(rdword, norun) \
{ \
    _asm push ebx \
    _asm push eax \
    _asm mov ebx, esp \
    _asm sub esp,rdword \
    _asm mov eax, fc_next_addr_fcb1 \
    _asm add esp, rdword \
    _asm push eax \
    _asm retn \
} \
norun \
fc_next_addr_fcb1: \
{ \
    _asm pop ebx \
    _asm pop eax \
}
```

这个宏含有 2 个参数。第一个参数是一个 word，你应该随意填写一个 16 字节数字，但是这个数字不要太大，比如 0x10。很容易看到下面的用法：sub esp,0x10 add esp 0x10,

这两句执行过后等于 esp 并没有改变。之所以用随机参数是为了让这段代码可以多种不同的形式出现，不会容易被找到规律一次性清除。

第二个参数是一段指令。这段指令永远不会被执行到，所以可以随意填写，之后代码跳到 fc_next_addr_fcb1 继续执行。前面压入堆栈的两个数据现在恢复，以便不影响后面的执行。这段宏可以加入正常函数的任意部位，每次都应该使用不同的参数。IDA 分析的时候，函数的整体性会被搅乱，一些代码甚至无法正常反汇编，而变成了简单数据。

请注意这只是一个例子。花指令的编写追求的是随机性和不确定性，绝对没有固定的准则，一旦形成固定的准则，则很快被反工程者熟悉，从而轻易去掉。如果你坚持只用上面提到的这个写法，阅读者经历一到两次迷惑之后，马上就会清醒过来。

关于 norun 这个参数，可以填入任何东西，但是最好不要是 C 语言语句，因为 C 语言语句可能在编译的时候被优化掉。我比较喜欢填入任意个数的 nop 或者 int 3，这样看上去更像是一个真正的内核函数的结束。

16.3.2 插入有意义的花指令

基础知识

花指令的另一个问题是：花指令一般是没有实际意义的，因此去掉并不影响其他的程序，所以能够被去掉。但是实际上，你也可以让花指令有其实际的意义。举个简单的例子：

```
xor eax,eax
```

这句指令大家都很熟悉，等于是 eax = 0。实际上 eax 清零有很多的方法，有些方法甚至比较难以看出是在清零 eax。比如，如下的写法：

```
push 0x63
mov eax, dword [esp]
sub eax,0x60
and eax,0x10      ;或者这里 xor eax,0x03 也是一样
… ;到这里 eax 被清 0 了，继续做正常的事，注意堆栈不平衡
pop eax ;结束之前恢复堆栈
```

其实很简单，如果要将 eax 清零，可以先令 eax = 0x63，然后减 0x60，再 and eax,0x10。这等于是本来可以一步到位的工作，做了一系列令人眼花缭乱的假动作来迷惑阅读者。

具有实际功能的花指令——也许已经不能叫做花指令了，很难被去除，因为它们包含了实际功能。

这个和刘涛涛的“扭曲变换”的思想近似。他的“扭曲变换”就是将现有的指令，用与之等效的但是更艰涩难懂且更多（当然也更没效率）的指令来替换。只不过他替换的是 obj 文件中已经编译好的指令，而我们没有那么高深的技术：研究 obj 文件的结构、机器码指令的解读那实在太困难了，还要做出正确的替换……本书直接在写 C 代码的时候加一些混淆来起到类似的效果，当然效果不可能有真正的“扭曲变换”那么强大。

这样的花指令（如果仍然叫做花指令的话）不能被整体去除，因为很显然，如果去除了，那么原本实际的功能也没有了。

代码示例

现在的问题是如何把上述功能变成宏。我们要的一般不是清零，而是赋值工作。赋值一般是往指定地址写数值。有许多方法可以写入数据，最常用的当然是通过寄存器 mov，但是实际上，stos、push、pop，甚至比较交换之类的指令都可以写入数据（可能最终结合寄存器并 mov）。此外，sub、add、xor 都可以对这些数字进行“无害”的计算。一些前述的“无意义的”花指令可以起到“隔离”作用，使有效的语句之间被分隔开。

```
// 赋值方法混淆代码段，本段主要用 xor
#define DZ_SET_1(address,data,rand_data,can_run) \
{ \
    unsigned int imd_data = (unsigned int)data ^ rand_data; \  \
    { \
        _asm push eax \
        _asm push imd_data \
        _asm mov eax,[esp] \
        _asm add esp,0x04 \
    } \
    can_run; \
    { \
        _asm xor eax,rand_data \
        _asm mov address,eax \
        _asm pop eax \
    } \
}
```

以上的宏是一个有效的赋值宏。对应的 C 语言是：

```
address = data;
```

rand_data 是随机数据；can_run 是任意一段可以执行的花指令。你可以直接嵌套前面我们写的 DZ_CAN_RUN1 宏。

上机实践

现在我们开始做一个实例。以下的内核程序，可谓世界上最简单的内核程序。同时，它也是极为标准的，几乎每个内核程序都有这样的操作：一个主入口函数、一个分发处理函数。在入口函数中设置分发函数：

```
NTSTATUS MyDispatch(PDEVICE_OBJECT dev, PIRP irp)
{
    return STATUS_UNSUCCESSFUL;
}

NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    size_t i;
    for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        driver->MajorFunction[i] = MyDispatch;
    }
    return STATUS_UNSUCCESSFUL;
}
```

这样的代码，反汇编的结果是清晰明了的。结果如下：

```
DriverEntry proc near
arg_0   = dword ptr  8
    push    edi
    mov     edi, [esp+arg_0]
    push    1Ch
    add     edi, 38h
    pop     ecx
    mov     eax, offset MyDispatch
    rep stosd
    mov     eax, 0C0000001h
    pop     edi
    retn    8
DriverEntry endp
```

这样的结果是反工程人士的最爱。可以清晰地看到：

（1）取参数 arg_0。

（2）然后循环变量赋 1Ch。

（3）给 arg_0 指向的地址后第 38 字节开始写入数据。

现在我只改动一点点。利用了上面两个宏：DZ_CAN_RUN_1 和 DZ_SET_1，代码修改成这样的：请注意只修改了 DriverEntry，没有修改 MyDispatch。修改多少代码只取决于哪些代码有保密的需求。

在这里读者可以深入体会那些宏嵌套的威力。

```
NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg)
{
    size_t i;
    size_t count;
    void *disp;
    // 这一句等于 count = IRP_MJ_MAXIMUM_FUNCTION;
    DZ_SET_1(count,
        IRP_MJ_MAXIMUM_FUNCTION,
        0x32ab,DZ_NOP_4);

    for (i = 0; i <= count; i++)
    {
        // 这句等于 disp = MyDispatch
        DZ_SET_1(disp,
            MyDispatch,
            0x3323,
            DZ_CAN_RUN_1(
                0x983a,
                DZ_NOPINT(3,5)));
        driver->MajorFunction[i] = (PDRIVER_DISPATCH)disp;
    }
    return STATUS_UNSUCCESSFUL;
}
```

这里的 DZ_NOP_4 表示 4 条 nop 指令。DZ_NOPINT(3,5)表示 3 条 nop 指令和 5 条 int 3 指令。这里用了三重宏嵌套。这样的代码也只能用在 DriverEntry 里（DriverEntry 只执行一次，即使慢点也没有什么影响。而阅读者一般都从 DriverEntry 开始阅读）；否则显得效率太低。

思考与练习

最后的反汇编代码……噢，我才不愿意去读它们，请读者自己反汇编试试。值得注意的是，一个显然的常数 1Ch（也就是 IRP_MJ_MAXIMUM_FUNCTION）将不会直接出现在反汇编代码中，但是另一个常数（38h，也就是 MajorFunction 在 driver 中的位置）依然直接出现。实际上，driver->MajorFunction 这样的地址意味着一个常数的出现（读者请思考一下这是为什么？）。你可以写一个类似的宏来掩盖这个数字吗？有兴趣的读者可以自己试试，作为本书的最后一个练习题。

第 17 章　用 VMProtect 保护代码

17.1　安装 VMProtect

基础知识

第 16 章“手写指令保护代码”中介绍的是将简单的指令变复杂的方法，需要手写大量代码，而且安全性和手写代码的质量极为相关。当然，对这样的扭曲手段，寻找快捷的逆向方法是不现实的，因此可以达到非常高的安全系数。但是一般在只需要满足比较简单的代码安全要求的情况下，大多数开发者没有那么多闲工夫去做这个，利用现成的工具对已经编译好的产品进行处理，依然不失为一个好方法。

只是所有现成的工具的不安全性在于：如果有工具可以处理代码，那么理论上就可以做出相反的工具：把处理过的代码还原。

比较传统的方案是：把原来的代码全部加密处理，保存在文件的某处。而驱动执行，则是先执行一段解密程序，把加密的驱动代码解密到内存里，然后再执行。

问题是，驱动以明码的方式在内存中执行，很容易用工具直接把内存中的内容全部 Dump 出来再分析，那就和没有加密一样了。

后来人们又发明了很多新的手段，比如把代码分段加密，每次只解密一小段，这一小段放入内存中执行。由于每次只执行一小段，那么把整个程序的全部代码都 Dump 出来就没那么容易了。

当然这中间涉及一些深层次的技术，比如有分支和跳转的代码。一个解密块正在执行，这时忽然发生一个跳转，要求跳转到某个地址。显然这个地址多半是无效的，因为这个程序并不是按原来的方式执行的，大部分代码还是加密作为数据放在内存里的。这个跳转要跳转到哪里去呢？

此时一般会根据所谓的基本块（BasicBlock）进行处理。一个基本块中间是没有跳转出指令，也不可能被跳转入的。基本块的跳转出只可能发生在最后一条指令，而被跳转到则只可能发生在开头。可以想象，无论一个程序多复杂，总是可以被分为若干个基本块。基本块可以作为加密的单位。如果一个基本块执行到之后发生了跳转，处理程序只需要知道这个跳转是跳转到另外的哪个基本块就可以了。这是可以预先就计算好的。

VMProtect 的思路又有所不同。VMProtect 翻译成中文是虚拟机执行保护，VM 指

Virtual Machine，也就是虚拟机。它的原理是把部分代码进行处理，使之变成非 x86 指令。这些指令当然不能直接在 x86CPU 上执行，但是却能在 VMProtect 的虚拟机上执行。实际上，这个虚拟机可以理解成一个小的解释程序，把处理后的新型指令解释成 x86 指令执行。但是这不是简单的加密。虚拟机可以有一套完整的指令系统，从虚拟机到 x86 指令之间的转换，可以说是一种编译或者解释的关系，而不是简单的加密处理的关系。这样一来，想破解的话，等于完整破解一套未公开的指令系统。没有简单的解密程序可以依赖，只能人工理解并进行翻译；或者自己写出 VMProtect 的逆向引擎。这些都不是不可能的，但是麻烦大大地增加了。

上机实践

VMProtect 的正式版本是要收费的，在进行商业开发的时候，请先购买授权。读者在学习此书时，可以先下载试用版本进行学习。VMProtect 1.22 是功能比较完善的试用版，更新的版本作者加强了试用版的限制，反而不如 1.22 版本的功能完整了。在下面的网页上应该可以找到链接：

http://www.pediy.com/tools/packers.htm

如果相关链接已经失效，请读者自己搜索“VMProtect”，或者与 VMProtect 的作者联系（请访问其官方网站 http://www.polytech.ural.ru/）。

VMProtect 1.22 的软件包非常简单，不需要安装，解压之后就可以直接执行 exe 来使用。实际上，笔者计算机上的 VMProtect 目录下只有 3 个文件，如图 17-1 所示。

图 17-1

蓝色图标的可执行程序是一个窗口应用，直接执行这个程序就可以了。第一次执行之后，初始界面如图 17-2 所示。

图 17-2

要处理一个可执行文件内核，请选择“File”菜单，再选择“Open”子菜单，然后选择一个已经编译好的 SYS 文件。具体的使用方法在 17.2 节中介绍。

这个软件一样可以处理应用层 PE 格式的文件，比如 EXE 和 DLL，方法也是一样的。但是本书以内核编程为主要内容，所以不针对性地进行介绍了。

17.2　使用 VMProtect

基础知识

VMProtect 并不是一次性对整个文件进行处理，而是以“过程”（Procedure）为单位进行处理的。实际上，读者可以把一个 Procedure 看成一个函数，虽然，VMProtect 的过程是可以比函数更小的单位。

其实这样更加方便。一方面，整个文件全部处理，显然意味着更多的开销。另一方面，文件可能会变得更大，而且运行起来也会更慢。在大多数情况下，程序中最重要的部分最需要加密，而且那样的地方并不是很多。

由于 VMProtect 处理的是已经编译好的 SYS 文件，那显然不像普通地写代码一样，使用者很清楚哪部分是要加密的。使用者必须指出 SYS 文件中的哪些部分是重要的，并告诉 VMProtect。这不是件简单的事。不过 VMProtect 提供了非常好的机制，让使用者在自己代码中插入标记，以便 VMProtect 能很轻易地找到那些重要的部分。

不过，笔者不喜欢修改自己的代码，所以用了更草率的方式：用 IDA 看自己编译的 SYS，找到重要的要加密的函数的地址，然后手工在 VMProtect 中指定来处理这些段落，使之转换为难以识别的 VMProtect 虚拟机指令。下面用实例来说明这一操作过程。

上机实践

按上一节介绍的方法，打开一个 SYS 文件之后，读者就可以看到在 VMProtect 的右边出现了一个多栏窗口。具体的截图如图 17-3 所示。

其中的“Dump”页面，显示的是整个 SYS 文件的反汇编结果，读者可以在这中间寻找自己想要加密的函数。不过糟糕的是，这个反汇编阅读显然非常的不便，光是从最

前面拖拉到最后就已经非常的麻烦，更不要说从程序入口点开始阅读了。

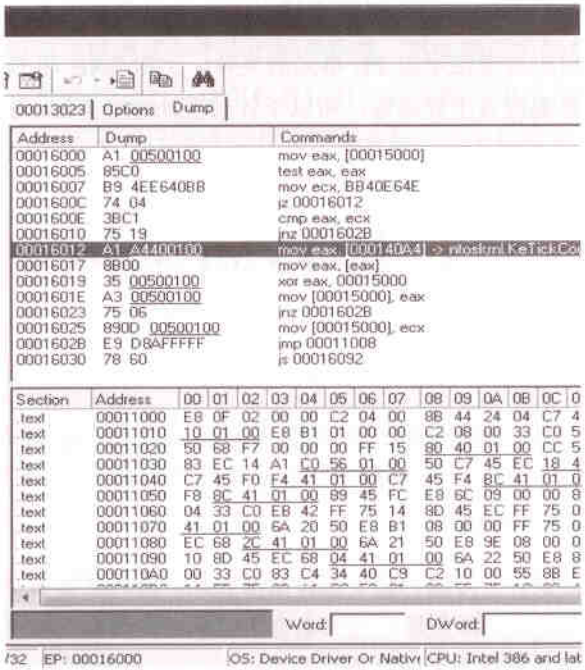


图 17-3

在代码中没有做标记的前提下，最好和 IDA 结合使用。下面是实际的例子。读者在学习第 14 章“反病毒、木马实例开发”的时候，应该已经开发了一个内核程序。如果还没有完成，请读者随意找一个简单的驱动编译出来（比如前面提及的 WDK 下的例子 diskperf）。

考虑到该实例用到的技术是内核 Hook。具体地说，是 Hook 了 ZwCreateSection 这个函数。这当然是极其常见和古老的方法，并不具备什么有价值的技术信息。出于保护技术的假设，我们假设：Hook 了 ZwCreateSection 这一信息是最关键的。如果本公司的竞争对手了解了这个技术点，则他们就可以做出类似的竞争产品。至于其他的部分，是公开的现实，大家都可以做。那第一个任务是找到这个关键点所在的位置。

代码分析

在源代码中，这个位置是很清楚的，但是在 SYS 文件中的位置则需要一些手段。现在打开 IDA，反汇编这个自己编译出来的驱动，入口点是这样的：

```
INIT:00016000                public start
INIT:00016000 start          proc near
INIT:00016000
INIT:00016000 arg_0          = dword ptr  4
INIT:00016000
INIT:00016000 ; FUNCTION CHUNK AT .text:00011008 SIZE 00000013 BYTES
INIT:00016000
INIT:00016000                mov     eax, dword_15000
INIT:00016005                test    eax, eax
INIT:00016007                mov     ecx, 0BB40E64Eh
INIT:0001600C                jz      short loc_16012
INIT:0001600E                cmp     eax, ecx
INIT:00016010                jnz     short loc_1602B
INIT:00016012
INIT:00016012 loc_16012:                             ; CODE XREF: start+C↑j
INIT:00016012                mov     eax, ds:KeTickCount
INIT:00016017                mov     eax, [eax]
INIT:00016019                xor     eax, offset dword_15000
INIT:0001601E                mov     dword_15000, eax
INIT:00016023                jnz     short loc_1602B
INIT:00016025                mov     dword_15000, ecx
INIT:0001602B
INIT:0001602B loc_1602B:                              ; CODE XREF: start+10↑j
INIT:0001602B                                         ; start+23↑j
INIT:0001602B                jmp     loc_11008
INIT:0001602B start          endp
```

通过前面的学习，读者应该已经很熟悉这段开头了。下面的 jmp loc_11008 直接跳转到 DriverEntry。在这里就不要一条一条地分析指令了，赶紧找到关键位置是最重要的。下面是 loc_11008。

```
.text:00011008 loc_11008:
.text:00011008               mov     eax, [esp+arg_0]
.text:0001100C               mov     dword ptr [eax+34h], offset loc_11000
.text:00011013               call    sub_111C9
.text:00011018               retn    8
```

loc_11008 简单得有点过分。这是因为作者在编写 DriverEntry 的时候，实际上只简单地调用了另一个初始化函数，可以看到就是 sub_111C9。那么马上看 sub_111C9 如下：

```
.text:000111C9 sub_111C9     proc near
.text:000111C9               push    esi
.text:000111CA               xor     esi, esi
.text:000111CC               push    esi             ; int
```

```
.text:000111CD                push    esi             ; int
.text:000111CE                push    esi             ; int
.text:000111CF                push    1F4h            ; int
.text:000111D4                push    1B7740h         ; int
.text:000111D9                push    7D0h            ; int
.text:000111DE                push    esi             ; int
.text:000111DF                push    esi             ; int
.text:000111E0                push    offset loc_111C3 ; int
.text:000111E5                push    offset loc_111A0 ; int
.text:000111EA                call    sub_1177F
.text:000111EF                cmp     eax, esi
.text:000111F1                mov     dword_156C0, eax
.text:000111F6                pop     esi
.text:000111F7                jnz     short loc_111FF
.text:000111F9                mov     eax, 0C000009Ah
.text:000111FE                retn
.text:000111FF ; ---------------------------------------
.text:000111FF
.text:000111FF loc_111FF:
.text:000111FF                push    offset sub_110F8
.text:00011204                push    offset sub_110AB
.text:00011209                push    offset sub_1102D
.text:0001120E                call    sub_11542
.text:00011213                retn
.text:00011213 sub_111C9      endp
```

sub_111C9 稍微有点复杂。但是对读者来说，显然读者应该有自己分析的模块的C语言代码，理解起来就应该极其快捷了。实际上，Hook ZwCreateSection 的功能是在 sub_11542 中完成的。sub_11542 的代码如下：

```
.text:00011542 sub_11542      proc near    ; CODE XREF: sub_111C9+45↓p
.text:00011542
.text:00011542 arg_0          = dword ptr  4
.text:00011542 arg_4          = dword ptr  8
.text:00011542 arg_8          = dword ptr  0Ch
.text:00011542
.text:00011542                mov     eax, [esp+arg_0]
.text:00011546                push    2
.text:00011548                mov     dword_156C8, eax
.text:0001154D                mov     eax, [esp+4+arg_4]
.text:00011551                push    2               ; int
.text:00011553                mov     dword_156CC, eax
.text:00011558                mov     eax, [esp+8+arg_8]
.text:0001155C                push    offset sub_11391 ; int
```

```
.text:00011561                push    offset aZwcreatesectio
; "ZwCreateSection"
.text:00011566                mov     dword_156D0, eax
.text:0001156B                call    sub_13023
.text:00011570                mov     dword_156C4, eax
.text:00011575                xor     eax, eax
.text:00011577                retn    0Ch
.text:00011577 sub_11542      endp ;
```

因为还没有进行任何代码保护处理，所以这里直接就看见了“ZwCreateSection”的字符串。下面调用函数 sub_13023 对这个函数进行了 Hook，这个函数也就是关键所在了。看完之后读者一定很吃惊——哇，这真是真相大白的一章。

```
.text:00013023 ; int __stdcall sub_13023(PCWSTR SourceString,int,int)
.text:00013023 sub_13023      proc near
.text:00013023
.text:00013023 SystemRoutineName= UNICODE_STRING ptr -8
.text:00013023 SourceString    = dword ptr  8
.text:00013023 arg_4           = dword ptr  0Ch
.text:00013023 arg_8           = dword ptr  10h
.text:00013023
.text:00013023                push    ebp
.text:00013024                mov     ebp, esp
.text:00013026                push    ecx
.text:00013027                push    ecx
.text:00013028                push    [ebp+SourceString] ; SourceString
.text:0001302B                lea     eax, [ebp+SystemRoutineName]
.text:0001302E                push    eax             ; DestinationString
.text:0001302F                call    ds:RtlInitUnicodeString
.text:00013035                lea     eax, [ebp+SystemRoutineName]

;IDA 甚至指出，这个参数是 SystemRoutineName，也就是要 Hook 的函数的名字
.text:00013038                push    eax             ; SystemRoutineName

;下面调用 MmGetSystemRoutineAddress，获得要 Hook 的函数的地址
;非常老套的做法
.text:00013039                call    ds:MmGetSystemRoutineAddress
.text:0001303F                test    eax, eax
.text:00013041                jz      short locret_13081
.text:00013043                mov     eax, [eax+1]
.text:00013046                mov     edx, eax

;连 KeServiceDescriptorTable 都出现了，无人会怀疑这里要做 Hook
;下面已经不用再读了，谁都能猜出来
```

```
.text:00013048                mov     eax, ds:KeServiceDescriptorTable
.text:0001304D                mov     eax, [eax]
.text:0001304F                push    esi
.text:00013050                push    edi
.text:00013051                mov     edi, [ebp+arg_8]
.text:00013054                shl     edx, 2
.text:00013057                mov     esi, [edx+eax]
.text:0001305A                push    edi
.text:0001305B                lea     eax, [ebp+SourceString]
.text:0001305E                push    eax
.text:0001305F                call    sub_12FD4
.text:00013064                mov     eax, ds:KeServiceDescriptorTable
.text:00013069                mov     eax, [eax]
.text:0001306B                mov     ecx, [ebp+arg_4]
.text:0001306E                add     edi, 0FFFFFFFEh
.text:00013071                push    edi
.text:00013072                mov     [edx+eax], ecx
.text:00013075                push    [ebp+SourceString]
.text:00013078                call    sub_13003
.text:0001307D                pop     edi
.text:0001307E                mov     eax, esi
.text:00013080                pop     esi
.text:00013081
.text:00013081 locret_13081:                  ; CODE XREF: sub_13023+1E↑j
.text:00013081                leave
.text:00013082                retn    10h
.text:00013082 sub_13023      endp
```

下面来处理这个函数吧。注意函数的开始地址“text:00013023”，注意 00013023 这个数字即可。在前面 VMProtect 的“Dump”页面里，这个地址也被标出来了。拖动滚动条到那个地址，执行操作如图 17-4 所示。

请在“Dump”中选中 00013023（当然读者分析自己的 SYS 文件，请自己找到其地址），然后在“Procedure”菜单下选择“New Procedure”，出现了询问框询问是否新的 Procedure 开始的地址是 00013023。单击“OK”按钮之后，左边就出现了一个 Procedure。选择 Tab00013023 可以看到这个 Procedure 下所有的代码（处理前的原始代码），一般地就是我们要处理的函数了。读者可以检查一下代码的对应情况。

读者可以依次操作，处理更多的函数，增加更多的 Procedure。然后点击工具栏上的类似播放的绿色小三角形，“编译过程”就开始了。一个新的 SYS 文件生成在和原 SYS 文件同一个目录下，一般名字为“原名.VMP.SYS”。请读者自己检查这个文件是否存在。

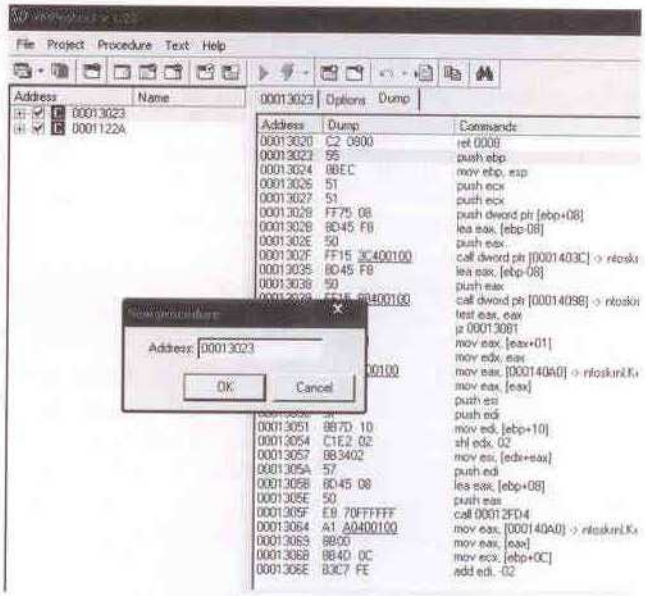


图 17-4

17.3　查看 VMProtect 效果

除了前面的简单介绍之外，笔者的技术水平很难清楚地阐述 VMProtect 的工作原理细节，毕竟我们不是职业的进行逆向工程的研究者。国内对 VMProtect 的专业研究也许最好的见于看雪论坛，有兴趣的读者可以访问：

http://www.pediy.com/

虽然不打算深入研究它的实现原理（一般研究这个的目的也就是为了破解 VMProtect 的保护），但是读者肯定很关心用 VMProtect 保护代码的效果。它到底有没有起作用呢？

现在请找到那个加了.VMP 标记的已经处理过的 SYS 文件。检查的方法很简单，用 IDA 再反汇编它，看看和原来的结果有什么不一样。

实际上，读者会发现从 start，DriverEntry 执行的那些代码，一直到处理过的函数之前为止，代码都没什么变化。还记得前面提到的 sub_11542 吗？连这个编号都没有改变。反汇编结果如下：

```
.text:00011542 sub_11542      proc near    ; CODE XREF: sub_111C9+45↑p
.text:00011542
.text:00011542 arg_0          = dword ptr  4
.text:00011542 arg_4          = dword ptr  8
.text:00011542 arg_8          = dword ptr  0Ch
.text:00011542
.text:00011542                mov     eax, [esp+arg_0]
.text:00011546                push    2
.text:00011548                mov     dword_156C8, eax
.text:0001154D                mov     eax, [esp+4+arg_4]
.text:00011551                push    2
.text:00011553                mov     dword_156CC, eax
.text:00011558                mov     eax, [esp+8+arg_8]
.text:0001155C                push    offset sub_11391
.text:00011561                push    offset aZwcreatesectio ;
"ZwCreateSection"
.text:00011566                mov     dword_156D0, eax
.text:0001156B                call    sub_13023
.text:00011570                mov     dword_156C4, eax
.text:00011575                xor     eax, eax
.text:00011577                retn    0Ch
.text:00011577 sub_11542      endp ; sp = -10h
```

处理过的代码是从 sub_13023 开始的。那么，应该顺这里开始检查，点开 sub_13023，果然山河剧变了，现在的 sub_13023 如下：

```
.text:00013023 sub_13023      proc near    ; CODE XREF: sub_11214+31E↑p
.text:00013023                                          ; sub_11542+29↑p
.text:00013023
.text:00013023 var_2C         = dword ptr -2Ch
.text:00013023 var_4          = dword ptr -4
.text:00013023
.text:00013023                push    0E1715A9Dh
.text:00013028                jmp     loc_1EB6F
.text:00013028 sub_13023      endp
```

这里是往堆栈中 push 了一个数字 0E1715A9Dh，然后便跳转到 loc_1EB6F 了。这个数字的意图还是不清楚，有兴趣的读者可以继续看：

```
.vmp1:0001EB6F loc_1EB6F:
.vmp1:0001EB6F                push    offset dword_1EAE5
.vmp1:0001EB74                jmp     loc_1D81A
.vmp1:0001EB74 ; END OF FUNCTION CHUNK FOR sub_13023
.vmp1:0001EB74 ; ---------------------------------------
.vmp1:0001EB79  dd  0B768CC08h,  0E930BF94h,  0FFFFFFDCh,  0C90F9C22h,
```

```
8CDF181h
    .vmp1:0001EB79  dd 435C8D5h, 48B534F6h, 0C0C1D1F7h, 2434810Ah, 84h,
0F740D6F7h
    .vmp1:0001EB79  dd 0FD8F7D9h, 0C1D5F7C9h, 0F74A1CCDh,
0F7D0F7D5h, 8FC181D3h
    .vmp1:0001EB79  dd 0C155BB28h, 8D4D08C8h, 0CB30A980h, 0FD2F723h,
81CF0FCAh
    .vmp1:0001EB79  dd 0CBF13DF3h, 80C281C6h, 0F7EB6182h, 4EDBF7D3h,
0AAECEF81h
    .vmp1:0001EB79  dd 0DEF74AF8h, 3DB2F281h, 0F7467890h, 19CFC1DFh,
3C5FEF81h
    .vmp1:0001EB79  dd 0F681C362h, 7D03CE3Ch, 0BC08C39Dh, 96CF683Bh,
0D8E99495h
    .vmp1:0001EB79  dd 3FFFFF95h, 32B668AAh, 0D8F7489Ch, 0D2F7D1F7h,
0A1952F35h
    .vmp1:0001EB79  dd 81D0F7DEh, 147CA7E9h, 81D3F7C6h, 64902F3h, 0F1F181B1h
    .vmp1:0001EB79  dd 0F7642B9h, 81424DC9h, 18B529F2h, 928D4E04h, 422ED13Dh
    .vmp1:0001EB79  dd 2434814Bh, 881h, 5D4AF781h, 0CE0FA64Eh, 0F03C3C1h,
8DDBF7CDh
    .vmp1:0001EB79  dd 0B305549Bh, 0F7DFF741h, 0FDDF7D3h, 0C1CD0FCBh,
0D7F70FC6h
```

前面又是一个类似的简单跳转，后面接着的是无数的 dd。本书作者的技术水平很难解释那些 dd 里隐藏着什么，也许是 VMProtect 自身解释需要的代码被隐藏在里面。

loc_1D81A 跳转到 VMProtect 自身执行的代码。这就不是本书所要解释的范围之内了，有兴趣的读者访问看雪论坛寻找前人的研究。

但是效果真的很不错，不是吗？以前真相大白的函数，现在令人摸不着头脑了。

除了 VMProtect 之外，还有其他的工具可以选择，比如 Virtualizer，请读者自己选择最合适的工具。当然读者需要记住的是：可以公开得到的工具，都只能提供有限的安全性。公开的工具提供的安全手段，是有许多研究者都可以破解的。

参 考 文 献

[1] 计算机组织与体系结构：性能设计（第 7 版）（世界著名计算机教材精选）

作者：斯托林斯、William Stallings、张昆藏、Kuncang Zhang

[2] Intel 微处理器

作者：Barry B.Brey，翻译：金惠华、艾明晶、尚利宏等

[3] Intel Architecture Software Developer Manual

Intel 公司

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
Broadview®
www.broadview.com.cn

《天书夜读——从汇编语言到 Windows 内核编程》

读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1. 短信

您只需编写如下短信：B07339+本书代码+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2. 电子邮件

您可以发邮件至**jsj@phei.com.cn**或**editor@broadview.com.cn**。

3. 信件

您可以写信至如下地址：北京万寿路**173**信箱博文视点，邮编：**100036**。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（**www.broadview.com.cn**）上，或推荐至**CSDN.NET**等专业网站上发表，被发表的书评的作者将获得价值**50**元的博文视点图书奖励。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱　博文视点（100036）　　电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明

《Windows环境下32位汇编语言程序设计（第2版）》

作者：罗云彬
ISBN:7-121-02260-5
出版时间：2006年03月

●畅销书升级版！

Windows环境下32位汇编语言是一种全新的编程语言。它使用与C++语言相同的API接口，不仅可以用来开发出大型的软件，而且是了解操作系统运行细节的最佳方式。本书尝试从编写应用程序的角度，从“Hello World”这个简单的例子开始到编写多线程、注册表和网络通信等复杂的程序，通过70多个从简单到复杂的例子，逐步深入Win32汇编语言编程的方方面面。本书作者有十多年的汇编语言编程经验，很清楚初学者在哪些地方会遇到问题，所以本书在系统全面地介绍Win32汇编编程的同时，也穿插了很多作者的经验之谈，使读者能够快速入门并最终熟练地写出各种Windows应用程序。

作者简介

罗云彬，软件工程师，现从事软件项目管理、软件工程实施、数据库应用等领域的工作，在Windows操作系统下的应用软件编程方面有丰富的经验，另外对Oracle数据库的管理有深入的研究，是国内为数不多的OCM证书获得者之一。

汇编语言编程是作者的一大爱好，作者自1990年开始即使用汇编语言编写程序，是Windows操作系统流行后国内最早研究Win32汇编编程的程序员之一，在1998年创建了专门探讨汇编编程的网站——http://asm.yeah.net，曾发表过大量关于汇编编程的文章和网上教程。

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

天书夜读—从汇编语言到Windows内核编程

我作为驱动开发的老兵，深感资料缺乏的艰辛，很多信息无法在文档中找到，此时自力更生的能力更加重要。

当我们手中拿着神兵利器——WinDbg时，一切都在掌握之中。这本书将要告诉您的不是全面的汇编知识或未公开的Windows秘密，而是怎么从这些貌似天书的汇编代码中，一探Windows底层的核心实现。

在开发中出现的问题，能不能从Windows自身找到答案？如果您正在这样思考，无疑本书是为您度身订做的。

本书授人以渔，也授人以鱼；短小精悍，读之如一缕清风，读罢则有醍醐灌顶之感。

——驱动开发网站长 马勇(znsoft)

作者介绍

谭文：从2002年到2008年，从事信息安全类软件的Windows内核驱动的开发工作。从2008年开始参与一个主要涉及不同架构之间二进制指令的实时翻译技术的项目的开发。业余时间在驱动开发网（www.driverdevelop.com）以楚狂人为笔名发表了许多技术文章。

邵坚磊：长期致力于x86体系架构与Windows系统底层技术的研究与相关商业软件的开发。目前在从事一个企业信息防泄密软件的Windows内核驱动的开发工作。是著名的反rootkit工具DarkSpy的作者之一。网名wowocock。

上架建议：汇编语言>Windows内核编程

ISBN 978-7-121-07339-7

9 787121 073397 >

定价：45.00元

网上订购：www.dearbook.com.cn
第二书店·第一服务

责任编辑：葛 娜
责任美编：谢丹丹